NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION OF A PROPOSED
SYSTEM FOR AUTOMATED
MICROCODE GENERATION

by

Marcia Elaine Provance

December 1984

Thesis Advisor: A. A. Ross

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Implementation of a Proposed System for Automated Microcode Generation

5. TYPE OF REPORT & PERIOD COVERED
Masters Thesis
December 1984

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Marcia Elaine Provance

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA. 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA. 93943

12. REPORT DATE
December 1984

13. NUMBER OF PAGES
128

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Microprogramming, Functional Design, Computer Control Units, Digital Implementation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis develops an automated microprogramming system. This system is designed around the goals of usefulness, usability, and security. The problem of mutually-dependent fields in a vertically formatted microinstruction is addressed, and a solution to this problem is presented. The proposed microprogramming system is organized around a series of menus which are presented to a microprogrammer so that she can build

microroutines by working on each microinstruction at a high abstract level.

Approved for public release; distribution unlimited

Implementation of a Proposed System for Automated Microcode Generation

by

Marcia Elaine Provance
Lieutenant, United States Navy
A.B., Pennsylvania State University, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL

December 1984

ABSTRACT

This thesis develops an automated microprogramming system. This system is designed around the goals of usefulness, usability, and security. The problem of mutually-dependent fields in a vertically formatted microinstruction is addressed, and a solution to this problem is presented. The proposed microprogramming system is organized around a series of menus which are presented to a microprogrammer so that she can build microroutines by working on each microinstruction at a high abstract level.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank two people for their special assistance during my studies at the Naval Postgraduate School. The first is Capt. Brad Mercer, USAF, who first introduced me to the background theory found in this thesis. I thank him for his great classes and for reading my thesis. The second person is Dr. Paula Strawser who taught me how to microcode, offered software engineering ideas during the design phase of this thesis, and also offered her comments on the final version.

I. DESIGN APPROACHES TO COMPUTER CONTROL UNITS

The discipline of Computer Science has evolved as the result of repeatedly applying two approaches to the solution of problems. The first approach is the decomposition of the entire problem or application into small, more manageable pieces; the second approach is to find a simpler algorithm for the application.

The decomposition of a problem can be done with two methods. The first is to see the application as a series of levels: The top level provides an abstract explanation of the application, and each lower level explains the application with an increasing amount of detail and complexity. The second method is to divide the application into separate components and to analyze each component in increasing detail.

An example of the division of a system into separate components is the traditional decomposition of a von Neumann digital computer into the five sections of control, arithmetic and logic, storage, input, and output. Each of these blocks can then be examined in detail or implemented in various ways which will not impact the other four remaining blocks. For example, the control block of a digital computer can be implemented using a hardwired configuration of gates and flip-flops or with a technique known as microprogramming. The implementation of a method of storage

or input/output operations will not be affected by the choice of control unit.

This example of control unit design can be extended to explain the second approach to achieving order and simplicity in digital systems. Microprogramming was originally developed as an attempt to find a regular and orderly hardware method to replace the jumbled mass of gates, flip-flops, and connections in a hardwired control unit. [Ref 1: p 159] Microprogramming should also improve a computer engineer's efficiency by providing an orderly and flexible design tool for the control block. An objective of this thesis will be to explore regular and ordered techniques for expressing microprograms. As a framework for the detailed description of microprogramming, the next section describes the control unit of a von Neumann digital computer and its hardwired implementation.

A. DESIGN OF HARDWIRED CONTROL UNITS

A digital computer, from the user's point of view, is a problem-solving machine. The user supplies input data, a switch is thrown, and output is produced. A more concrete view is held by the computer engineer who sees the system as an elaborate array of interconnected flip-flops and logic gates which transfers information around the system. [Ref 2: p 4] A computer scientist's view of a digital system is a combination of the abstract and concrete views. She knows

that the computer consists of hardware structures made from the engineer's flip-flops, gates, and logic paths; however, the computer scientist also realizes that the purpose of the computer is to interpret and execute user-written instructions which will access user-provided data in order to solve the stated problem. The responsibility of directing this problem solution belongs to the control unit. This job can be described as information transfer among the five functional units of the computer. This information transfer will decide which instructions to execute, what data to use as operands, and which hardware components to activate. The control unit communicates with control signals which choose the correct data path, and it activates specific logic gates and flip-flops. [Ref. 3: p 52]

Information transfer can best be explained by analyzing the instruction interpretation and execution cycle of a stored program computer. A sample program and hardware configuration will be used to assist the explanation. The example user problem is to add a constant 2 to a 2 stored at a memory location and store the result back into memory. In assembly and machine language for a hypothetical machine, the instructions and their direct addresses in main memory might be as follows:

ADDR	Assembly Inst	Machine Inst
Program storage		
000	LDI 002	001 010
001	ADD 004	010 100
010	STR 005	011 101

```
    011          STP 000                100 000
Data storage
    100          000 002                000              010
    101          000 000                000              000
```

This program will load the constant 2 into the accumulator, add to the accumulator the contents of memory location 004, store the resultant sum into memory location 005, and stop execution.

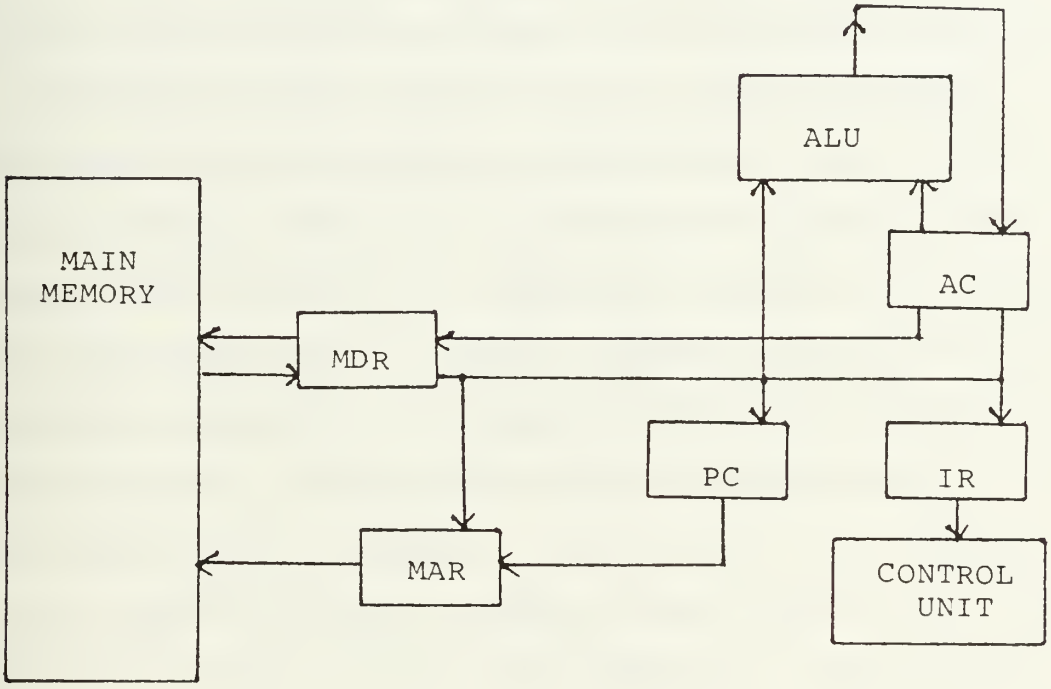

Figure 1 [Ref 5: p 269)]
Sample Hardware Configuration

The sample hardware configuration is found as Figure 1. The simple computer consists of a main memory, a control unit, an arithmetic and logic unit, and five special purpose registers. These registers are the instruction register

(IR) which holds the current instruction, the program counter (PC) which contains the address of the next instruction, the memory address register (MAR) which contains the location in memory to be accessed for a read or write operation, the memory data register (MDR) which will hold the data that has been read from or will be written to the main memory, and the accumulator register (ACC). At the start of execution, all the registers are cleared to 0.

An instruction can be viewed as a request to the control unit to generate control signals which activate specific data paths so that information can move among the functional units and between the registers. The control signals also activate the arithmetic and logic unit (ALU) so that desired functions will be performed. [Ref. 2: p 4] The instruction interpretation and execution cycle will cause the correct signals to be generated in the correct order. The cycle can be decomposed into five steps: 1) fetch the instruction, 2) decode the instruction and increment the PC, 3) fetch the required operands, 4) perform the function, and 5) store the result. [Ref. 4: p 107]

In step 1, the contents of the PC are transferred to the MAR, and the contents of the memory location specified by the MAR flow from main memory through the MDR into the IR. These inter-register and inter-unit transfers can be expressed in a shorthand known as Register Transfer Language. One comment must first be made about the steps. Each

of the five steps in the instruction interpretation and execution cycle may require register transfers. The steps in the example refer to the instruction interpretation and execution cycle, while the T's refer to the clock pulse. In step 1, the following register transfers will take place:

Step	Pulse	Logical	Physical
1	T1	MAR <= [PC]	MAR <= 000
	T2	IR <= [[MAR]]	IR <=001 010

In step 2, the instruction to be executed is determined by decoding the left half of the instruction register. Each instruction in a digital computer's instruction set is identified by a unique pattern of bits. These bits are found in the left half of the IR, interpreted by the control unit, and instruction-specified signals are generated in steps 3, 4, and 5. In the case of the load-immediate instruction, the control unit knows that the operand is contained in the right half of the IR. If the instruction were a load from a memory location, the control unit would know that an address was contained in the right half of the IR and would generate those control signals which would generate a memory access. Also, the PC is incremented in this step.

Step	Pulse	Logical	Physical
2	T3	PC <=[PC] + 1	PC <= 001

In step 3, the operands are fetched and placed into the appropriate registers. For the load immediate instruction,

the constant 2 is placed into the ACC. Step 4, perform the function, and step 5, store the result, do nothing for this particular instruction.

Step	Pulse	Logical	Physical
3	T4	ACC <= [IR(right)]	ACC <= 010

Interpretation and execution of the ADD instruction is done in the same manner.

Step	Pulse	Logical	Physical
1	T1	MAR <= [PC]	MAR <= 001
	T2	IR <= [[MAR]]	IR <= 010 100
2	T3	PC <= [PC] + 1	PC <= 010
3	T4	MAR <= [IR(right)]	MAR <= 100
	T5	MDR <= [[MAR]]	MDR <= 000 010
4	T6	ACC <= [ACC] + [MDR]	ACC <= 010 + 010

The third instruction, the store, is interpreted and executed as follows:

Step	Pulse	Logical	Physical
1	T1	MAR <= [PC]	MAR <= 011
	T2	IR <= [[MAR]]	IR <= 011 101
2	T3	PC <= [PC] + 1	PC <= 011
3	T4	MDR <= [ACC]	MDR <= 100
	T5	MAR <= [IR(right)]	MAR <= 101
4	T6	enable write signal	
5	T7	[[MAR]] <= [ACC]	[101] <= 100

The control unit of a digital computer is concerned with the transfer of information by generating control signals in

the order specified by the above cycle. These control signals in the proper sequence effect the interpretation and execution of user-provided instructions. It should be noted that the first two steps for every instruction are the same; this is the interpretation portion of the cycle. Mainly, this cycle changes a static machine into a dynamic problem-solver. Two techniques have been applied in the design of control units so that this transformation can be made; they are hardwired control units and microprogrammed control units.

Hayes describes hardwired control units as those that use fixed logic circuits to interpret instructions and generate control signals. There are three possible design approaches for this type of control unit: 1) the sequential circuit design of switching theory with the construction of a state table for the control unit, 2) a method based on the use of delay elements for control signal timing, and 3) a method that uses sequence counters for timing purposes. [Ref. 5: p 245] Patterson also provides a description of hardwired control. In a hardwired control system, a network of electronic logic is devised that will recognize each object code instruction in the computer's instruction set; each object code instruction is a pattern of signals which are sent to the control unit. This network decodes the instruction. The control system will transform these

signals into another set of unique signals which will effect the opening and closing of gates on the selected data paths. [Ref. 3: p 52]

A third description of a hardwired control unit is as an assemblage of interconnected combinational and sequential networks that function as a finite state machine. [Ref. 6: p 3] Hayes' state table approach would be used for this control unit. The main points about hardwired control units to be remembered are the unique nature of the pattern of bits for each instruction and the instruction-unique control signals which are generated after decoding the object language instruction.

Hardwired control units are designed in an ad hoc manner with the computer designer reducing logic equations and drawing block diagrams until a satisfactory arrangement is found that meets the cost, schedule, and performance requirements. The process of deriving the equations and their logical implementation will be described. First, all of the control signals which need to be generated to implement all the machine language instructions in the computer's repertoire are listed. Examples of some of these are PC_{out}, MAR_{in}, Read, Write, MDR_{out}, and END.
Multiple combinations of the following three items will be listed for each control signal belonging to the target digital computer being designed: Each instruction which required that specific control signal for interpretation

or execution; the step of the cycle where the control signal must be generated; and the presence and state of condition codes necessary for signal activation. Our example will be the control signal for the end of a program. The END control signal will be generated for the instructions which require it, within the specified clock cycle, and with the testing of the needed condition code. The logic equation of an END is END = T_8 * ADD + T_7 * BR + (T_7 * N = T_4 * BRN + ...

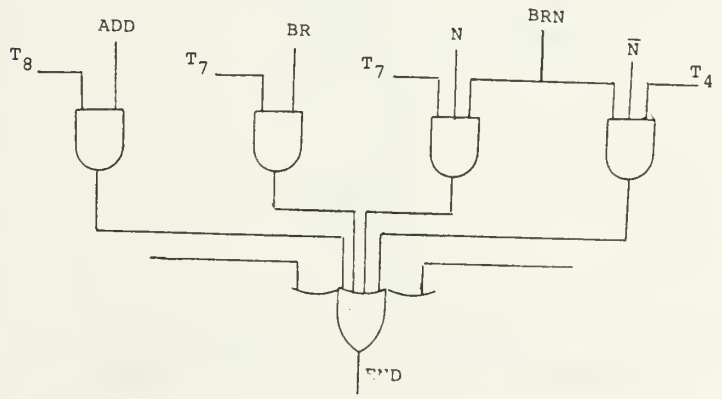

Figure 2 [Ref. 4: p 113]
Implementation of Logic Equation

The physical implementation of the above equation is found as Figure 2. The equations and their physical implementations are completed for every control signal. All of these independently-designed logical implementations are placed into the control unit. A hardwired control unit is shown as Figure 3.

The ad hoc construction of the encoder and decorder results in complexity which will increase in proportion to the size and completeness of the machine language instruction set. An unmanageable and confused tangle of gates and interconnections often results from the minimizations of the logic equations and the ad hoc combinations and uses of gates, flip-flops, their interconnections, and the size of

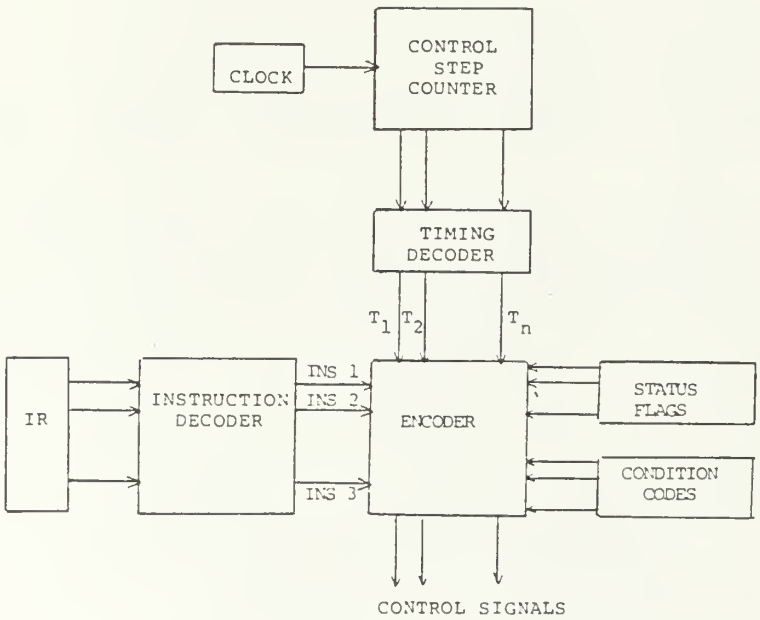

Figure 3 [Ref 4: p 112]
Hardwired Control Unit

the instruction set. The resulting hardwired control units are difficult to test and maintain since the control unit has no order or regularity. Changes are also expensive

since, most often, the entire control unit must be redesigned and replaced. The desire to incorporate order, modularity, flexibility, and maintainability in control unit design leads to the development of a different type of control unit.

B. DESIGN OF MICROPROGRAMMED CONTROL UNITS

In 1949, Professor Maurice V. Wilkes of the University of Cambridge set out to find a better way to organize the control functions of a digital computer system. At that time, Wilkes invented the method of control unit design known as microprogramming. Wilkes' design goal was to eliminate the randomness of control logic and replace it with an orderly logic matrix. The concept of microprogramming makes it easier to understand the control function and to build hardware because it replaces the complex circuitry with a repetitive, ordered array of memory cells. In addition to reducing complexity, microprogramming gives digital systems new flexibility; the control flow can be changed without redesigning the hardware. [Ref. 3: p 54]

The best illustrations of what microprogramming really is and how it works come from the original work published by Maurice Wilkes. His description begins with definitions. The operation called for by a single machine instruction can be broken down into a sequence of more elementary operations. These elementary operations are referred to as

micro-operations; examples of micro-operations are PC_{out}, MAR_{in}, and ACC_{in}. Basic machine operations like addition are made up of a microprogram of micro-operations. Those micro-operations which take place during the same clock pulse are placed into the same micro-instruction. The process of writing a microprogram is similar to writing an application program in machine language. [Ref. 1: p 158] This idea places microprogramming not only in the realm of hardware design but also into the areas of concern for software engineers. Consequently, concepts like information hiding and hierarchical, modular design can be used to advantage in microprogramming.

For microprogramming to work, certain hardware structures are required. The machine must contain a permanent rapid-access storage device which will hold the microprogram. Means are also required to determine and effect the sequencing or order of the microinstructions for both sequential and conditional control flow. A microprogrammed system consists of two parts. The first is the control register unit; this is a group of registers and the ALU together with a switching system which enables transfers to be made. The second part is the micro-control unit; its concern is to control the sequence of those micro-instructions required to carry out each object code instruction and to cause the proper control signals to be generated.

The micro-control unit is shown in Figure 4; it consists of a decoding tree, two random access memories, and two registers. A series of clock pulses will be generated and applied as an input to the decoding tree; the output activated from the tree depends upon the contents of register I. This action corresponds to step 2 of the instruction interpretation and execution cycle; this is how an object code instruction is decoded by the microprogram. The output line

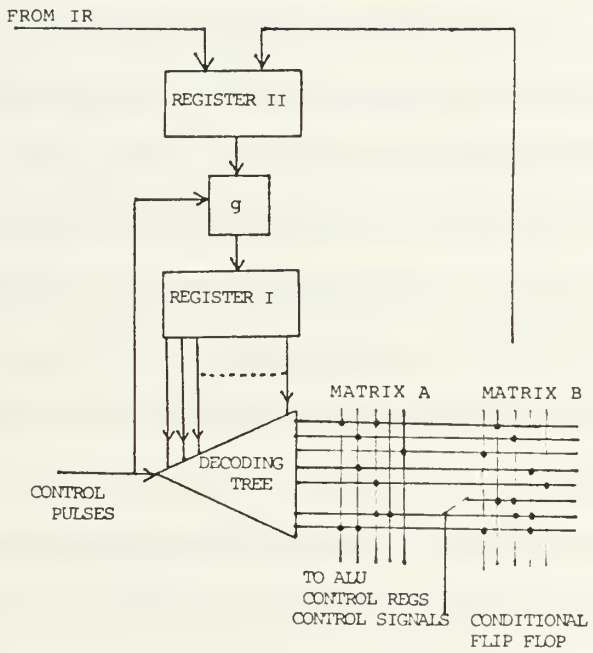

Figure 4 [Ref. 1: p 159]
Wilkes' Original Design of a Microprogrammed Control Unit

will contain the address within the random access memory which is the first micro-instruction of the microprogram for the object code instruction found in the IR. This output

line passes into the first random access memory, called a rectifier matrix by Prof. Wilkes. The outputs of this matrix are the control signals which operate the various gates and flip-flops associated with the micro-operations. The output lines of the decoding tree also pass into rectifier matrix B, and its outputs are connected to register II. The contents of register II are the address of the next micro-instruction to be executed. Before the control pulse is applied to the decoding tree, the contents of register II are transferred to register I. the decoding tree is now ready to provide Matrix A with the address of the next micro-instruction whose output will be the next set of control signals for the various hardware components. This application of clock pulses alternatively to the input of the tree and to the connection between register I and II causes the predetermined sequence of microinstructions to be executed. [Ref. 1; p 159]

A succinct description of the above process is provided by Hayes. Microprogramming is a method of control unit design where control signal selection and sequencing information are contained in a random access memory. The control signals which are to be activated at a particular time are specified by the micro-instruction which has been fetched from the control memory. Each microinstruction will also specify the address of the next microinstruction to be executed. [Ref. 5; p 271] Rauscher and Adams provide a

definition of microprogramming that relates it to an analysis of levels of abstraction. Microprograms contain information that control hardware at a primitive level, and these microprograms are stored in a special memory and sequenced as stored programs. A computer will be termed microprogrammed if the instructions which are directly fetched, decoded, and executed correspond to the primitive operations that the machine performs. [Ref. 7: p 4]

C. ADVANTAGES AND USES OF MICROPROGRAMMING

Since the inception of microprogramming in 1949, advances in memory technologies have provided advntages for control unit design and have provided many uses for microprogramming. Several sources point to the advantages of microprogramming as the design method for control units. The primary advantages are flexibility and maintainability. It is very easy to add a machine language instruction to an instruction set or to change the entire instruction set in a microprogrammed system. All that is required is for a new control store to be designed which will hold the improvements and to replace the old control store. All other circuitry and hardware within the computer system will not be affected. [Ref. 7: p 5; Ref. 2; p 72]

IBM was one of the first organizations to exploit the flexibility of microprogramming when it designed the System/360 family of computers. All of the family members deffered

in terms of internal hardware, organization, and structure; but each computer contained a comprehensive instruction set that could be used by any family member to interpret and execute machine language instructions. This idea by IBM started the use of a concept known as upward and downward compatibility.

Another exploitation of the flexibility of microprogramming is in the transformation of a general purpose computer into a specialized problem machine. In a hardwired computer, it is the responsibility of the programmer to tailor the system to solve her problem by using numerous general purpose instructions. If a specific problem needs to be solved many times, it can be placed in the hardware by being microprogrammed. With the use of a microprogrammed control unit, a microprogrammed subroutine would be implemented inside the control store as one microprogram with a single corresponding machine language instruction. The hardware would then better support the programming environment, and programmers would find programming a more efficient task.

Microprogrammed control units are also easier to develop and maintain. The substitution of simple, repetitive memory structures makes the design process easier. Also, concepts used in software engineering such as modularity, information hiding, and structured programming can be applied to the creation of microprograms. It is easier to maintain and improve microprograms since only the control store is

replaced, and the underlying computer organization is not affected. A computer can also respond more easily to new performance demands and problem solutions. A richer or a larger instruction set can be implemented, and a more responsive system is ready to start work.

Other advantages of microprogramming include changeability, economy, and ease of education. It is possible to have more than one instruction set resident in a single digital computer. It is also possible to allow for numerous architectural characteristics to be chosen and implemented. This is accomplished by having more than one control memory containing microprograms resident within the system. The programmer would be able to choose the hardware configuration or the instruction set which best matches the performance criteria of her problem. The economy of microprogramming is a result of its simplicity. Since there is less circuitry in a microprogrammed computer, less sequential logic will need to be procured in order to implement a rich and full instruction set. The systematic design approach taken for microprogrammed control units may also reflect a savings in design time. The simplicity and order in the internal circuitry of a microprogrammed machine and the methodical techniques used in its design make it easier to teach microprogramming to system designers and engineers. Flowcharts and microprograms written in symbolic languages are the tools for the microprogrammer; the

hardwired control unit designer will use sequencing and timing sheets in addition to complicated hardware logic sheets. The tools will be easier to teach. [Ref. 2: pp 72-75]

Rauscher and Adams provide an outstanding summary of the various uses of microprogramming. The first application is in emulation. With emulation, the instruction set of one computer is embedded into the control store of a different computer. The host computer will interpret and execute machine language instructions as the target machine would. One current use of this technique is the emulation of new architectures for research purposes. Another use of emulation is in a software first machine. During the acquisition phase for a computer system, different machines could be evaluated by loading their instruction set into a software first machine and running benchmark programs against each target computer.

A second application of microprogramming is in the area of operating systems. Current work in this area has two approaches. The first is to implement primitives that are used throughout the operating system as microprograms, and the second is to implement important portions of the operating system as microprograms. A third application of microprogramming is in the support of higher-level language programs. In this approach there can be many machine languages for each high level language. Each machine language would be targeted to a different performance criteria for

the high level language. A fourth development is the use of high-level microprogramming languages. In this application, a user's program would be written in a high-level language which would be continuously translated until the lowest level of language would be microcode. There would be no interaction between an object code program and microprograms. The last use of microprogramming pointed to its use in architecture implementations. Examples include pipeline structures, floating point processors, and multi- and distributed processing. [Ref. 7: pp 16-18]

II. MICROPROGRAMMING METHODS

A. DIFFICULITY OF MICROPROGRAMMING

Although Maurice Wilkes developed a more systematic method for control unit design, microprogramming wasn't used commercially unitl the early 1960s. During the 1950s, it was felt that any computer system which would use micro-programming as the implementation of a control unit would not meet the speed requirements in terms of instruction interpretation and execution times. When Wilkes conceived his different approach to control unit design, rapid access memories were not available. Advances in the semiconductor industry solved this problem, and a fast and cheap RAM was available by the early 1960s. As the amount of information

stored on a chip increased, the price declined, and rapid access to the microinstructions became possible. Microprogramming had become practical in terms of hardware. IBM was the first computer vendor to produce a family of microprogrammed computers. [Ref. 6: p 56] Since the early 1960s, several other computer vendors have developed microprogrammed digital computer systems; some of these vendors are Hewlett Packard, Digital Equipment Corporation, and Burroughs.

While microprogrammed control units were being implemented by major computer manufacturers, the task of creating the various microprograms was not easy. Microprogramming is a very labor-intensive task. A microprogrammer may spend hours just to optimize, by hand, 10 or 20 microinstructions. This time-consuming task has become infeasible when the current size of microprograms is considered. [Ref. 8: 702] Nothing was automated in the process of creating microcode; the microprogrammer worked at a very low level with a binary language. The opportunity for error was quite high, and the microprogrammer was forced to remember address and bit positions in their absolute terms instead of using memonics and symbolic labels. A first step toward automating the production of microcode was meta-assemblers, but they still have left many problems. These problems will be discussed in section C--Low Level Microprogramming.

The creation of large microprograms using high-level microprogramming languages is a current approach to the problem of creating large microprograms in a realistic time frame. In order to provide a context for the discussion of high-level microprogramming languages, a discussion of high-level languages and their impact on problem solution is presented next.

B. LEVELS OF ABSTRACTION IN PROGRAMMING LANGUAGES

A computer system and the problem that it will solve can be decomposed by viewing the system and the specific problem as levels of abstraction. This concept can best be explained by looking at the various classes of programming languages. The top-level abstract explanation of the problem can be done in a higher-level English like Pseudo-code. Then a high-level computer language like Pascal expresses the problem in English-like statements which are impossible for the computer to understand without further translation. The next level is the translation of the high-level language into an intermediate-level language which is closer to the type of language understood by the computer. This intermediate language is then translated a final time into object code. This object code is the lowest or next-lowest level of programming languages, depending upon how it is interpreted and executed. If the hardware structures which

interpret and execute the object code instructions are randomly-configured logic gates and flip-flops, the object code is the lowest level of decomposition. The object code may, however, be further interpreted and the program executed after interaction with another level of language known as the microprogram. This is the lowest-level statement of the problem but is a more general low-level language which interprets each object language instruction statement and activates various hardware structures in order to solve the target problem.

In the history of programming languages, the earliest programs were written in machine languages. High-level languages evolved as a method to make problem statement and solution easier for people to express and develop. These higher-level languages require compilers, assemblers, linkers, and loaders as translators. These translators introduce an overhead cost because of the interaction of the operating system, the system software, and the application program. Machine efficiency is reduced because of the various translations.

C. LOW-LEVEL MICROPROGRAMMING

Microcode was originally produced, much like machine code, with the microprogrammer working in a binary machine language. She would also be responsible for optimizing this

microcode by hand. No automated tools or microsystem software was available. The history of the development of microprogramming tools and languages parallels that of application programming languages. The first step was the production of meta-assemblers which introduced the use of mnemonics to microprogramming.

Meta-assemblers represent the bits associated with a particular field of the microinstruction with a mnemonic name. For example, a mnemonic for the bits representing the ALU Source fields might be ASOURCE. Creating a microprogram using a meta-assembler is a two-step process. The first step is to define the language in terms of the mnemonics and assign a bits(s) position to the mnemonic. As an example, a 48-bit microinstruction will be used. The last four bits indicate the flow of control within the microprogram. A binary 1110, or a hex E, indicates a continue to the next microinstruction. The mnemonic would be CONT and would represent a bit pattern of 1110 in bits 44-47. Part of creating the mnemonics is defining the structure of the microword. The length of the word is determined, and the various field meanings and representations are created. A second part of creating microcode using a meta-assembler is writing the actual microcode which solves the target problem. The example of the hardwired control unit design of adding two and two will be recreated here in the format used by a meta-assembler to illustrate this approach to creating microcode.

```
0.  NOOEY,RAMAB,NOP,RAM,,,,LDIR,RF,RF,CONT
1.  ,RAMAB,INC,RAM,CIONE,,,RF,RF,JMAP
2.  LDI: NOOEY,RAMAB,NOP,RAM,,,READIR,R1R1,JZ
3.  ADD: NOOEY,RAMAB,NOP,RAM,,,READIR,RA,RA,CONT
4.       NOOEY,RAMAB,NOP,RAM,,,Read,R2,R2,CONT
5.       OEY,RAMAB,ADD,RAM,CIZERO,,,R1,R2,JZ
6.  Store: NOOEY,RAMAB,NOP,RAM,,,READIR,RA,RA,CONT
           OEY,RAMAB,INCRS,YBUS,CIZERO,,RA,R2,JZ
7.  Stop: ,,,,,,,R8,CJP
```

This method of creating microcode using a meta-assembler has the advantage that some automation of code construction has occurred. The microprogrammer no longer is forced to remember which bits control which hardware structures; she may now use mnemonics which suggest the hardware function, and she is freed from having to remember how many bits determine the hardware function. This method is still error prone because the mnemonics are postion dependent. It would be easy to place the mnemonics out of order, misspell one of them, or to forget one of the commas. The microprogrammer is not totally freed from memorization because the mnemonics must be remembered or written down. Another point that requires mentioning is that there is a translation phase involved with this method--the mnemonics must be translated into microcode. The microprogrammer is still forced to state the problem at a very low level. A very good knowledge of the hardware structures, their control signals,

and the microprogramming language is required since design is done one low-level statement at a time.

D. HIGH LEVEL MICROPROGRAMMING LANGUAGES

Increased demand for systems and applications written in microcode suggests that a higher-level of abstraction may be required for microprogramming languages. Three developments point to this new requirement. The first is a change in the authors of systems written in microcode. Traditionally, computer architects were the only people who wrote microcode; now people outside of the architectural group, but still inside the company, need to write microcode for their systems. A common example is the designers of an operating system who want to implement certain speed-critical parts of their system in microcode. These people are interested in the speed benefits of microprogrammed systems, but they do not want to learn all the details of the machine which would be required if a meta-assembler were used. A higher level microprogramming language would enable a more abstract problem definition, and the operating systems' designers could more easily produce microcode. [Ref. 8: p 704] Another requirement for the use of levels of abstraction in microprogramming languages is the increasing complexity of computer architectures. The primary result of this is larger instruction sets for the macro-level machine language

which will cause more complex and larger microroutines. As an example, the PDP 11/70 used 256 microinstructions to implement the machine language while the VAX 11/780 requires more than 5000 microinstructions. [Ref. 8: p 704] The third demand for high-level microprogramming languages is the ability to tailor a computer system. Computer users want to realize the advantages of transforming a general-purpose problem solver into one with a specific architecture focused on their applications. The basic instruction set can be enlarged or optimized for a particular task. The ability to microprogram a system has made this type of refinement possible. A high-level microprogramming language will allow the users of a system to perform such tailoring in a reasonable timeframe [Ref. 2: p 57].

High-level microprogramming languages (HLML) should provide an increased measure of programmer efficiency similar to that of other high-level languages such as PASCAL. The hierarchical structures of HLML may make it easier to perform global optimizations which provide more of a speed efficiency than hand optimzations. [Ref. 8: p 704] David A. Patterson at the University of California, Berkeley, has developed an ALGOL-derived HLML named STRUM. The goal of his work was to determine the impact of modern programming techniques on microprogramming. [Ref. 8: p 700] He wished to demonstrate that a high-level language, structured programming, and program verification would improve the

correctness and efficiency of microprograms. Patterson felt that his research had produced an efficient high-level language. He first pointed out that the use of a HLML made the production of code easier. Secondly, the code is understandable, which is important from the maintenance point of view. [Ref. 8: p 704] Microcode is seldom maintained by the person who created the original version; thus readability is an important criteria for the code. STRUM also provided the level of abstraction desired by non-computer architects and required for describing complex computer architectures.

Another microprogramming language was also developed at the University of California, Berkeley, by David A. Patterson, Karl Lew, and Richard Tuck. Their goal with this language was to investigate the possibility of creating an efficient high-level microprogramming language that would be machine independent. Their first step in this direction was to produce a machine-independent low-level language which they named Yet Another Low Level Language (YALLL). [Ref 9: p 22] The creators of YALLL felt that this was a good first step beause it would not be difficult for a compiler to produce YALLL, and optimizers would be able to translate YALLL into efficient microcode. [Ref. 9: p 22] YALLL shared the cirteria of readability and understandability; these same features were found when comparing early macro-low-level languages with machine code. [Ref. 9: p 23]

It is necessary to look at both the advantages and disadvantages of high-level microprogramming languages. From the STRUM and YALLL studies, Patterson and his fellow researchers concluded that problem definition and solution were earier to write and understand in the higher-level microprogramming languages. This is a reasonable conclusion considering the precedent in application-directed high-level languages. The conclusion was also drawn that a problem definition and solution written and executed using a high-level microprogramming language would have a speed advantage over a problem definition written in a conventional programming language. A reason for this conclusion is that the final translated version of the high-level problem solution (the object code) would not have to interact with general-purpose microroutines to activate the hardware facilities. A last advantage of high-level microprogramming languages as seen by Patterson was that they optimized the microcode. In the research for both STRUM and YALLL, the resultant microcode was compared against microcode prepared for the same problem definition either with a meta-assembler or by hand. In the case of STRUM, the microcode produced was as efficient as that produced by hand. [Ref. 8: p 705] In the YALLL study, the code was comparable with that produced for one of the target computers. [Ref. 9: p 24]

E. CRITICISM OF HIGH-LEVEL MICROPROGRAMMING LANGUAGES

The practitioners of microprogramming have been slow to accept high-level microprogramming languages. It is the speed of the control unit which determines the speed of the problem solution. [Ref. 2: p 52] The main criticism about using microprogramming as the means to generate control signals is the time required to fetch and decode each micro-instruction before the control signals can be produced. [Ref. 4: p 25] Speed and efficiency of execution has been more of a concern with microprogramming languages than with application programming at the macro level since microcode is the language level closest to the hardware of the machine. The speed of problem solution directly depends upon the speed of microroutine interpretation and execution.

When high-level microprogramming languages are introduced for problem solution, this speed disadvantage is compounded. The primary uses of microprogramming are instruction set implementation, emulation, and speed-sensitive operating systems applications. These uses are not a direct utilization of microprogramming to solve a specific problem. In these cases, microprogramming is a tool used by the hardware and the systems software to accomplish general problem solution. Each reference to the microcode will involve the layers of decomposition associated with any high level programming language. The time penalty may be intolerrable. While the work accomplished with STRUM and YALLL by

Patterson is a sound approach to high-level application-specific problem solution, the tradeoff cannot be afforded. The user/writer of an application in a high-level microprogramming language must forego speed advantages at execution in order to make problem definition and solution easier to write and more understandable to read. When microprogramming is seen in the context of a tool used by the system, the speed requirement is paramount.

A last criticism addresses the knowledge required by the user of a high-level microprogramming language. This task requires a working knowledge of both language and compiler design, Backus-Naur Form for describing the grammar of the language, and an intimate familiarity with the hardware structures and their associated control signals. If a language like STRUM were available, the user of the system would still have to be familiar with the hardware in order to tailor STRUM to her specific application. STRUM is a machine-dependent high-level micro-programming language.

The approach of using a high-level microprogramming language to define and solve a target problem may not be suitable when considering the tradeoffs involved. Meta-assemblers are also undesirable because of the low level of detail at which the microprogrammer must work. A middle-ground solution is needed which allows the production of speed-efficient microcode but removes the drudgery from the

task of microprogramming. This thesis presents an automated system which allows the microprogrammer to work on each microinstruction at an abstract level and provides the mechanism to produce microroutines.

The implementation described in this thesis assumes that the microprogrammer has already created the algorithm to solve the target problem and expressed it in some pseudo-code. Each step in the pseudo-code algorithm is a summarization of an individual microinstruction. The microprogrammer is then ready to access the proposed system and prepare the algorithm as microcode in its final hex format. The system will present increasingly detailed menus beginning at the level of a series of microroutines and progressing to the actual fields within a specific micro-instruction. The end product of the system will be user-named microroutines of varying length constructed from microinstructions in the binary or hex lowest-level format required by the target architecture. Although the abstraction capability for the entire problem in a high-level language will not be realized with this system, the microprogrammer can still work at a high level with a microroutine; and she will realize advantages over the meta-assembler method. First, the microprogrammer is released from the drudgery of memorization; the meaning, order, and spelling of mnemonics are eliminated. Second, typographical errors will be reduced since fewer keystrokes are required.

The menu responses are only one character. Third, table lookups which are required for selecting the value of mnemonics or determining the exact meaning of a mnemonic are also eliminated because the tables are summarized and reproduced in the menus. Further error control is provided by automatically processing mutually-dependent fields. The microprogrammer does not have to remember the mutually-dependent fields or those fields whose meaning and use are determined by the choice made in another field. For example, if the function chosen for the ALU of a hypothetical machine restricts the possible ALU source operands, the microprogrammer will only be allowed to choose permissible sources.

The method proposed in this thesis for writing microcode is an improvement over methods currently in use. The method attempts to preserve the speed efficiency of microcode by producing code in its lowest level of abstraction while the microprogrammer is spared the traditional tedium of working at such a low level. The next chapter provides a detailed explanation of the proposed system.

III. PROPOSED MICROPROGRAMMING SYSTEM

A. GOALS OF THE SYSTEM

The proposed microprogramming system design was driven by the four goals of usefulness, usability, security, and general purpose application. The system would be considered useful if a microprogrammer would perfer to use it as opposed to using other microprogramming methods currently available. Another criterion for usefulness is the correctness of the microcode. If the microroutines created by a microprogrammer using the proposed system correctly and efficiently solved the target problem/application, then the design would be considered useful.

A comprehensive system is a last component of usefulness. A system must anticipate all the actions that a microprogrammer would need to make in order to build microroutines. These actions, at the level of the series of microroutines, are the ability to scan the names of all existing microroutines and print the microroutines. The microprogrammer is given the ability to name/create, find, list, add, and delete a specified microroutine. An existing microinstruction can be located based upon a key and then modified or deleted. New microinstructions can be inserted into an existing microroutine or added to the bottom of the Microroutine. The last action required by a microprogrammer

is the capability to have all the work done during the terminal session saved to a disk file or to build the system's data structure from a disk file at the start of a terminal session. A complete session will walk a microprogrammer through all the levels of abstraction from many micro-routines to a single field in a specific microinstruction. Once the microprogrammer makes a choice, the system should know the requisite order in which to present the menus. The mechanism is also needed which allows the microprogrammer to navigate the various levels, save or destroy all completed work, and terminate the session. A useful system provides all the actions that a micropro-grammer would reqire once no algorithm is complete.

The usability of the system refers to the man-machine interface provided by the system. This man-machine interface should allow an easy creation of microcode. First, the microprogrammer needs to be relieved of the requirements to memorize mnemonics and to refer to various references for required information about the microinstruction or the architecture. The menus summarize all tables and present comprehensive choices to the user. All that a microprogrammer should require to create microcode using this system is the detailed algorithm and the file name of the system. Secondly, the entry requirement needs to be reduced. With this system, the microprogrammer no longer enters mnemonics or the actual binary or hex values for the fields. All

entering is in response to menus, and all but one response are one character long.

The most important criterion of a usable system is that it replicate the process that is used to create microcode by hand. In this particular case, the order in which menus are presented should closely approximate the order in which the microprogrammer completes fields in the microinstruction when using a manual system. Basically, there is a one-to-one correspondence between a field in the microinstruction and the scope of each menu. The basis for replicating the manual process is ease of use. Microprogrammers tend to approach the fields of the microinstruction in the same order. If the system presents the same fields in the same order, the microprogrammer will find the system easy to learn and use.

The goal of security is motivated by a desire to eliminate errors made by microprogrammers. Security is not considered to mean protection of one microprogrammer's code from another microprogrammer. Security as defined in the scope of this thesis refers to protecting the microprogrammer from herself. No action made by the microprogrammer which violates the format or the contents of the microinstruction should go undetected. [Ref. 10: p 527] The reduction in keystrokes, memorization, and table lookups should eliminate some errors. The most important errors which need to be handled by the system are the interaction of mutually

dependent fields and subordinate fields. A microinstruction format is described as horizontal or vertical. In a horizontal microinstruction format, each field will have only one use or meaning. If the microinstruction format is vertical, all or some of the fields will have more than one use. For example, the same field may be used to hold a microstore branch address, a register selection value, or a constant value to be loaded into a counter. The exact meaning of this field will depend upon the exact value of other fields in the microinstruction. As a further extension of the vertical format example, suppose that the fields which interact are the ALU source field, the above described branch address field, and the sequencer control field. If the next microstore address is based on a conditional branch, the branch address field would contain a register selection if an ALU source operand is contained in that selected register. A field conflict will exist with shared fields. In the above hypothetical microinstruction, the next microinstruction address cannot be determined by a conditional branch if one of the ALU source operands is contained in a register. In a manual microprogramming system, the microprogrammer might not recognize and correct such a conflict. With the proposed system, the ALU source operand will be checked against the next microstore address to see if a conflict was present; if a discrepancy is present, the microprogrammer will be warned.

A last source of error which the proposed system attempts to prevent is subordinate fields. Dependent upon the choices made for the value of a field within the microinstruction, other fields within that same microinstruction or another microinstruction will need to be completed or contain a specific value. In a manual system, the microprogrammer must remember what these fields are and if any constraints are placed on the value that the field in question may hold. The proposed system will present the microprogrammer with the menus for the subordinate fields, and only the legal choices will be displayed for selection. If the fields affected are in another microinstruction, the user will be warned what range of values must be in the preceeding or succeeding microinstruction. It is the microprogrammer's responsibility to reference the preceeding word or remember the requirement for the succeeding microinstruction. Figure 5 shows the data path taken by the system when the microprogrammer is selecting the next microstore address source.

The last goal of general purpose applicability is difficult to implement when considering the various microprogrammable architectures and microinstruction formats. The top-level capability is to allow a microprogrammer to select any target machine and design her own microinstruction format in terms of length, field size, position, and hardware component controlled, and mutually

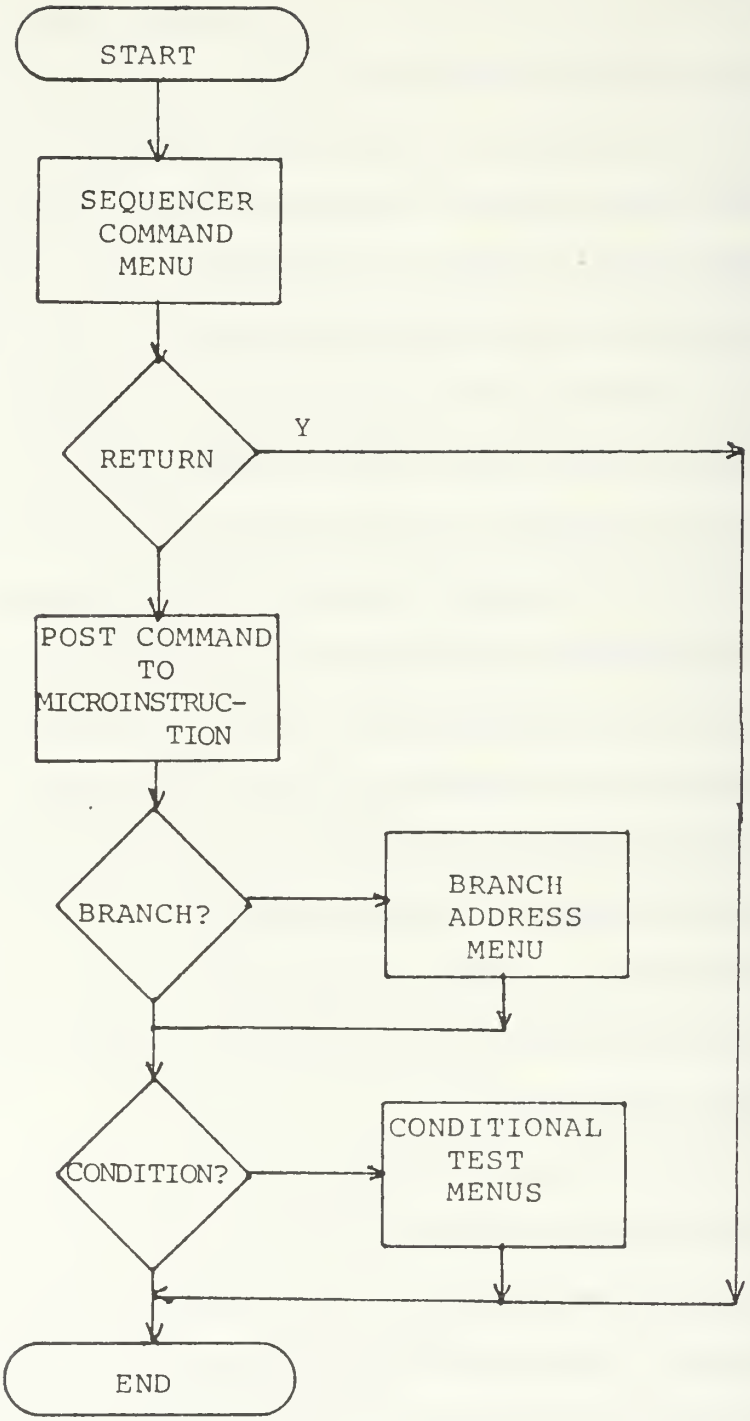

Figure 5
Next Microinstruction Address

dependent or subordinate fields. The proposed system does not meet this goal. In the history of programming languages, the original low level languages and even FORTRAN were machine dependent. [Ref. 10: p 41] The proposed system and its menus are predicated on a specific microprogrammable target machine and a fixed microinstruction format. This goal of retargetability is still important, and it must be considered as a primary goal for the next system designed to ease the task of microprogramming.

B. THE TARGET MICROPROGRAMMABLE MACHINE

1. The Am29203 Evaluation Board

In order to build a new technique for generating microcode and to test the new method, a target microprogrammable digital system is required. Available at the Naval Postgraduate School is a prototype of the AM29203 Evaluation Board. This board was initially designed for microprogramming experiments. It is built from various bipolar chips produced by Advanced Micro Devices in Sunnyvale, Ca. The chips used belong to the AM2900 family. The evaluation board is used only for explanation of the microprogramming technique created by the proposed system. Other microprogrammable systems are available and could also be used to demonstrate how this online microprogram generator functions.

The target microprogrammed system consists of three sections: computer control unit (CCU), the arithmetic and

logic unit (ALU), and the macro-level memory and I/O. The block diagram of the evaluation board is found as Figure 6.

The main hardware component of the CCU is the AM2910 Sequencer. This microprogram controller is an address sequencer for controlling the sequence of execution of microinstructions stored in microprogram memory. Both sequential access and conditional branching to any address in microprogram memory is provided. [Ref. 13: p 5-123] A diagram of the AM2910 microprogram controller is shown as Figure 7. The other hardware structures include a writable control store, a mapping PROM which translates an op code contained in the Instruction Register into an address in the writable control store, and a pipeline register and decoding PROM which increases the vertical microprogramming depth. [Ref. 12: 3-10]

The arithmetic and logic unit used in the target machine is the AM29203 four-bit microprocessor slice. This ALU chip can perform seven arithmetic and nine logical functions on two four-bit operands. AM29203s can be cascaded to provide for varying length operands. The evaluation board cascades four AM29203 ALUs to allow the handling of 16 bit operands. Sixteen special functions are also supported which facilitate division, multiplication, binary/BCD conversions, and mormalization. [Ref. 13: p 5-342] Figure 8 is a block diagram of the AM29203, and Figure 9 is a diagram of how the four AM29203s are connected on the evaluation board.

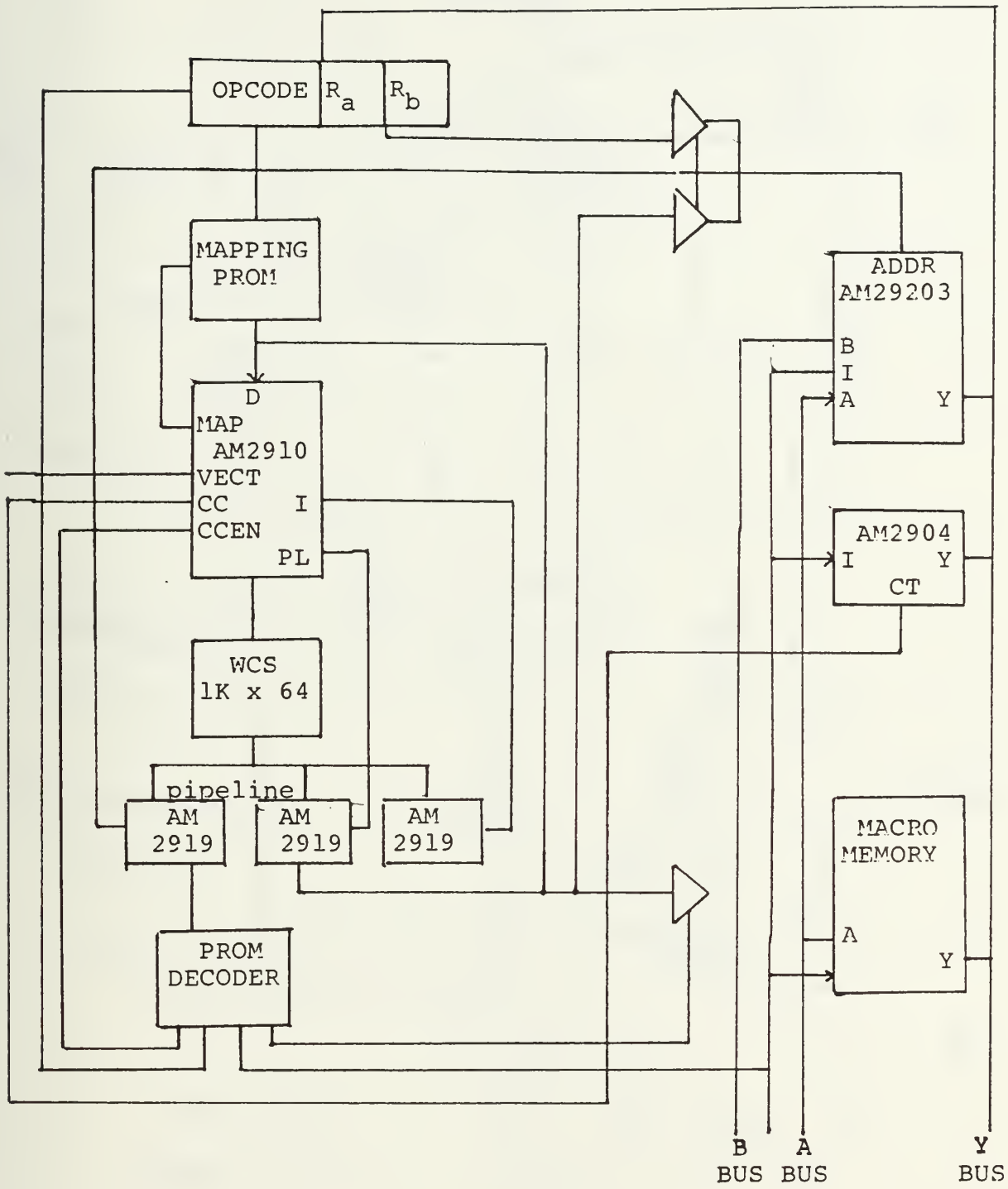

Figure 6 [Ref. 12: 9-2]
Block Diagram of AM29203 Evaluation Board

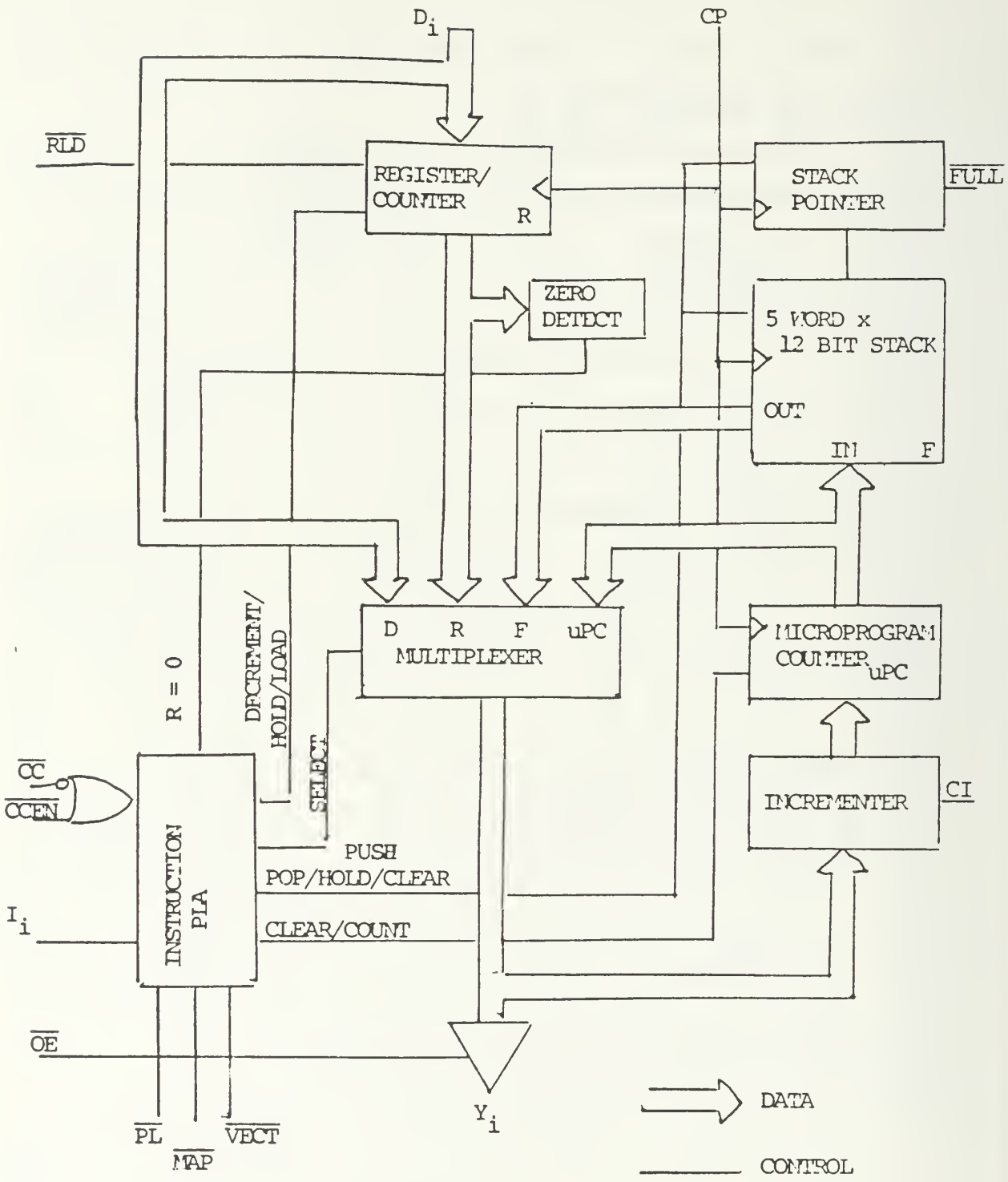

Figure 7 [Ref. 13 p. 2-12]
Block Diagram AM2910

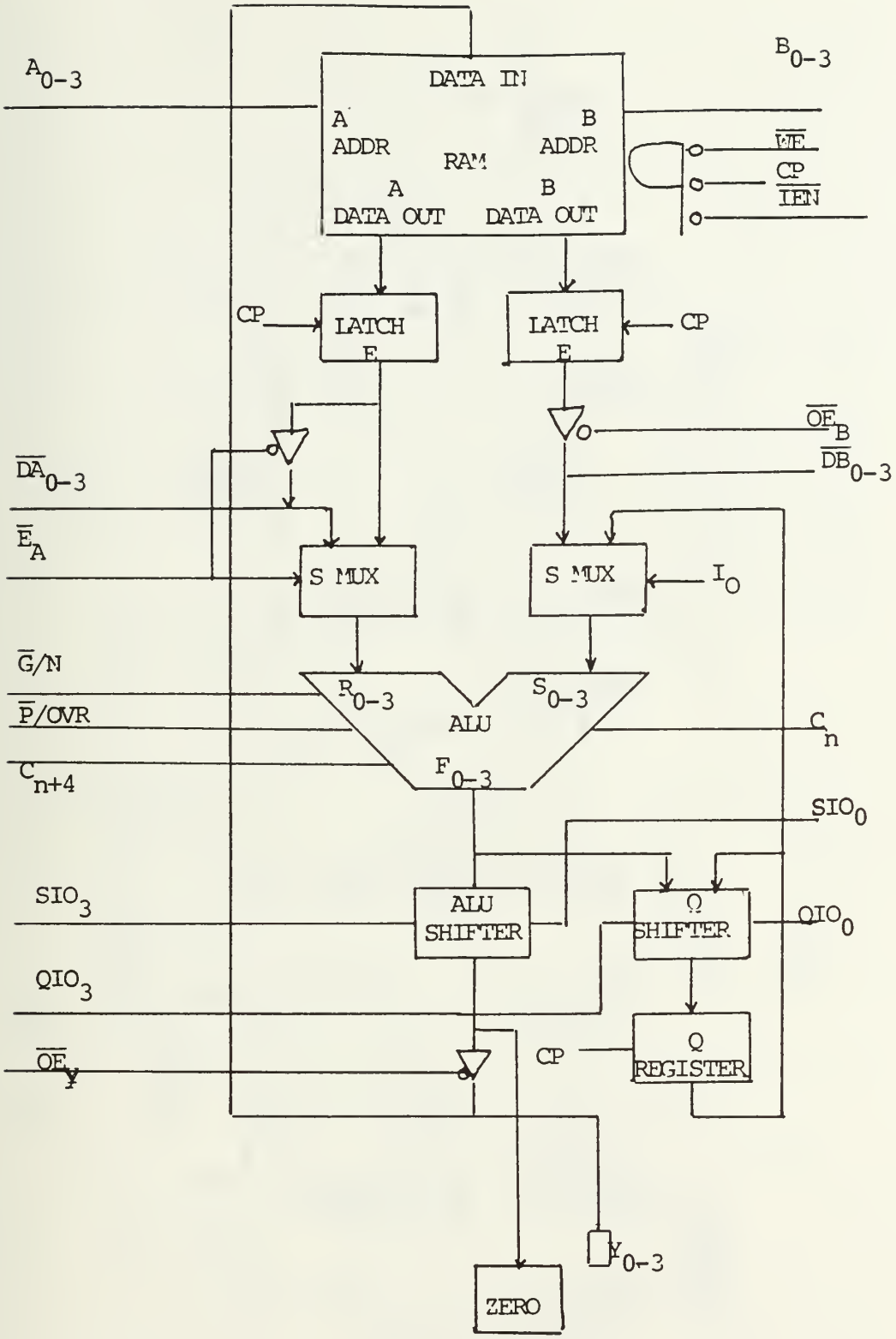

Figure 8 [Ref. 13: p. 2-16]
Block Diagram of AM2903

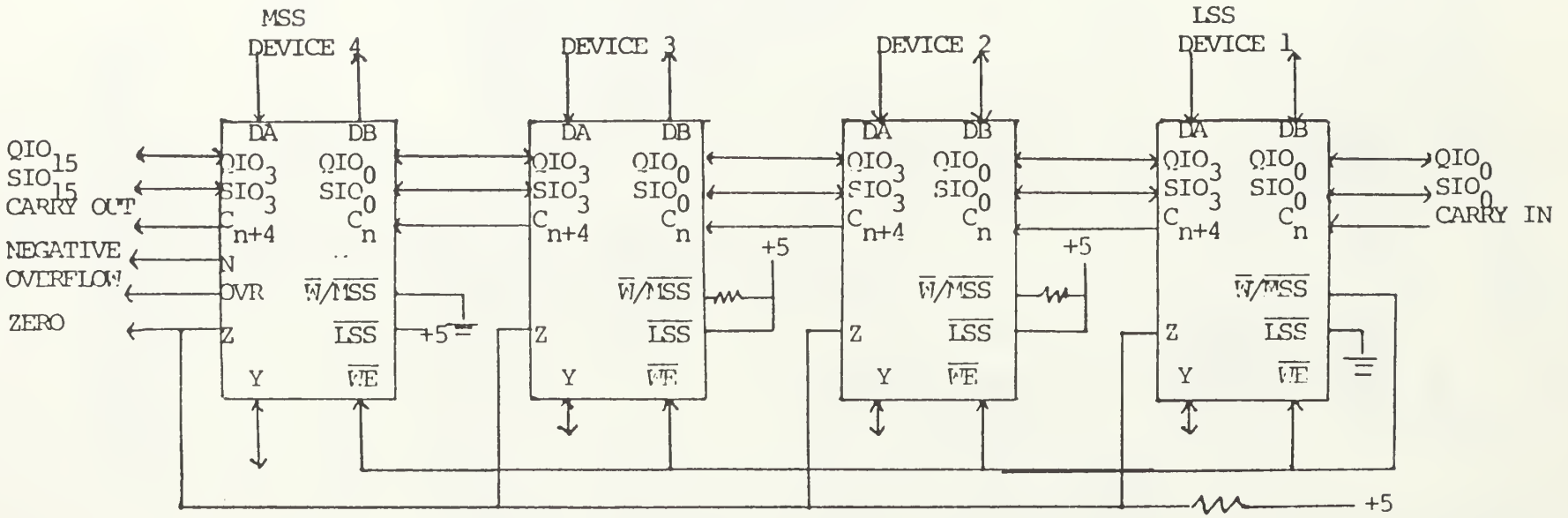

Figure 9 [Ref. 13, 7-8]
AM29203 on Evaluation Board

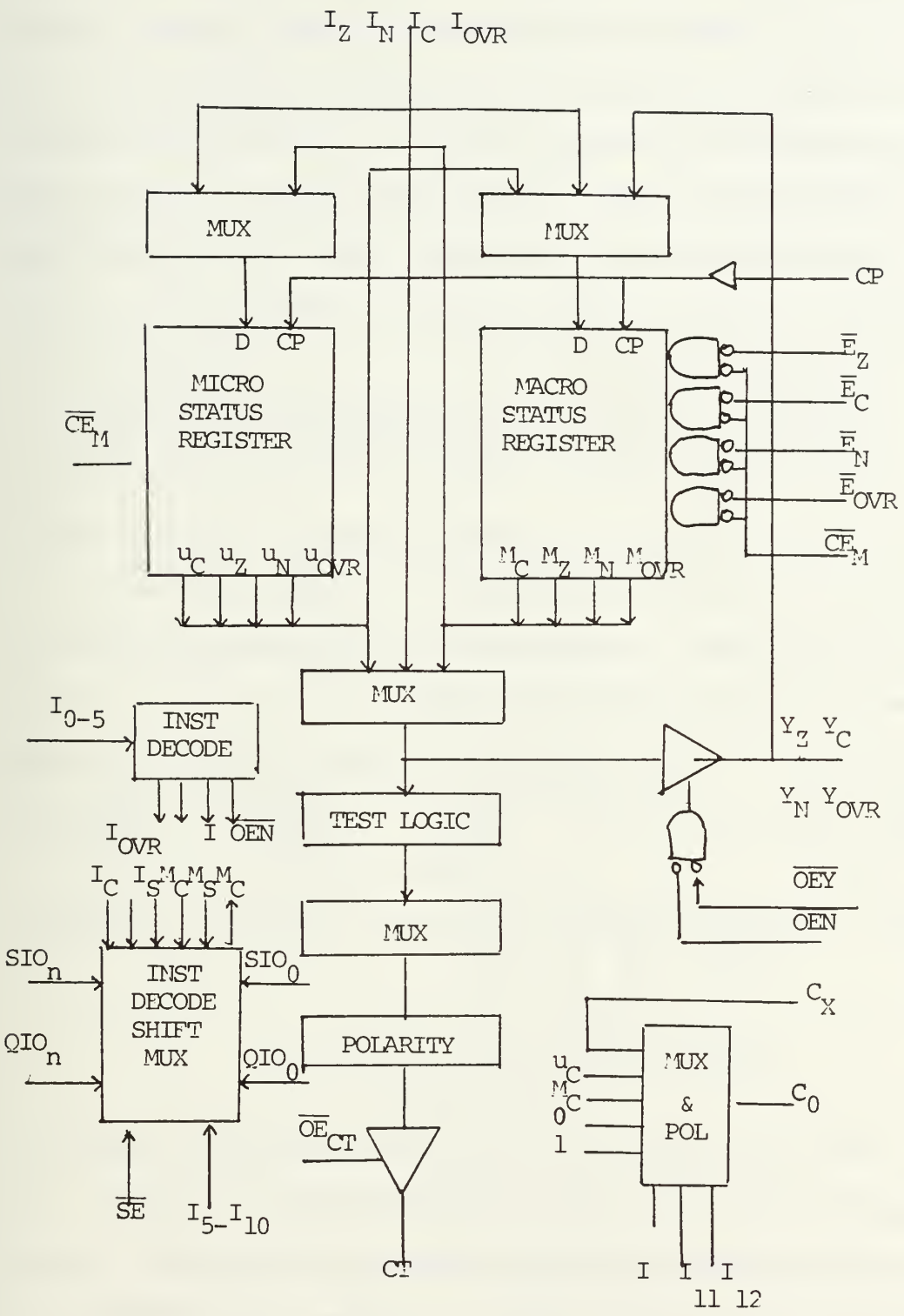

Figure 10 [Ref. 13: p. 8-2]
Block Diagram AM2904

The second major hardware component found in the ALU section is the AM2904 Status and Shift Control Unit. This integrated circuit performs the miscellaneous functions which are required to support an ALU. The AM2904 is three nearly independent blocks of logic which provide shift linkages, status registers and condition code checking, and the carry-in for the ALU. Figure 10 is a block diagram for the AM2904. [Ref. 13: p 5-72]

2. AM29203 Evaluation Board Microinstruction Format

The microinstruction consists of 48 bits which are organized into three major fields. A general microinstruction format is provided as figure 11. These three main groupings of fields correspond to the three main hardware components of the evaluation board.

OPERAND REGISTER ADDRESS	ALU OPERATIONS	CONDITION CODES	SHIFT AND CARRY	MICROIN-STRUCTION BRANCH	NEXT ADDRESS SELECT
AM 29203	AM 29203	AM 2904	AM 2904	AM 2910	AM 2910

Figure 11 [Ref. 12: p 3-5]
General Microinstruction Format

The first portion of the microinstruction controls the hardware associated with the AM29203 ALU. This part of the microinstruction is shown in detail in figure 12. The first three bits are the register address select fields which specify either the pipeline register in the CCU or the

Macro Instruction Register as the sources of ALU operands or the destination of an ALU operation. The next bit is the instruction enable which controls whether the result of an ALU operation is written to any of the ALU RAM registers if they are the selected destination. The next bit is also an enable which determines if the ALU output appears on the Y

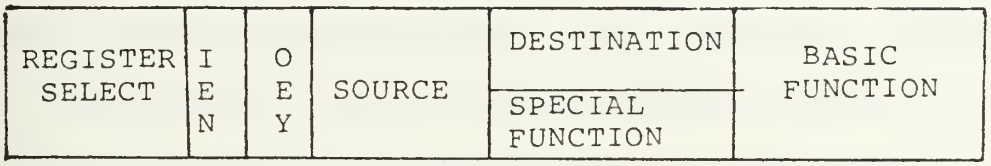

Figure 12
AN29293 ALU Portion of the Microinstruction

bus. The Y bus is the major data bus in the evaluation board. The last three fields are the ALU Source Operand selection, ALU destination selection, and ALU function selection.

Figure 13
AM2904 Shift/Status Control Portion of Microinstruction

Since the AM2904 chip performs the different functions of carry-in, status-checking, and the setting-up for conditional tests, many of the bits within the microinstruction

control different hardware structures. The first two bits of the AM2904 portion of the microinstruction control the carry-in when it is 1, 0, or the output of the ALU. The next six bits are the Instruction Lines for the AM2904 and are numbered I5-I0. These are the six bits that most strongly bring to light the problem of mutually-dependent fields. These bits control what is done to the micro and the macro status registers, the state of the carry-in if the carry-in's source is a status register or an immediate input, and the register and the condition reflected in a conditional test. A detailed discussion of these six bits and their associated problem of mutually-dependent fields is contained in the next chapter. The next two bits are the enable bits for the Macro Status Register and the micro status register. The last six bits are primarily concerned with the shift linkages required by the ALU special functions or ALU destination. These bits are also used by the board to enable communications off the board, to enable memory reads and writes, and to load the Instruction Register. The first bit in this section is the shift bit which enables or disables the shift linkages, the second bit is the command bit, and the last four bits help to uniquely determine the actual shift pattern or the miscellaneous and memory function to be performed by the system. The complete layout of the center sixteen bits of the microinstruction is provided as figure 13.

The last set of fields belong to the AM2910 sequencer whose format is illustrated in figure 14. The first two bits, bits 15 and 14, provide for a breakpoint in execution and a spare bit. Bits 13-4 are the multiple-purpose bits that were described in an earlier example. This Branch Address Field contains a branch address, the RAM register ipentifiers for an ALU operation, or a constant which can be loaded into a counter or register. The last four bits are the AM2910 sequencer command field which implement sequential or conditional flow of control within a microroutine.

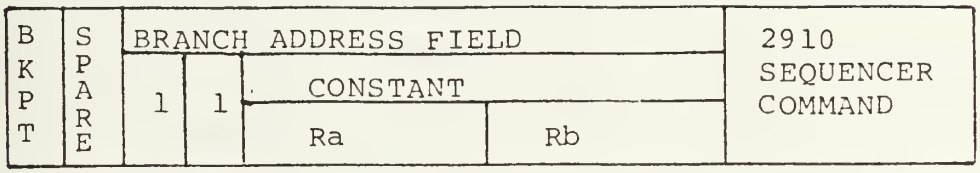

Figure 14
AM2910 Sequencer Portion of the Microinstruction

C. ENVIRONMENT OF THE SYSTEM

The proposed microprogramming system was coded in Berkely Pascal, and it is run on a VAX 11/780 computer under the control of the UNIX operating system. It is an interactive program which presents the microprogrammer with a series of menus. There are various paths through the system depending upon the choices made by the microprogrammer. The microprogrammer can proceed in both directions through the hierarchy of the system.

The data structure used by the program is a linked list which contains two types of records. This structure allows the user flexibility when performing operations on microroutines and microinstructions. With a linked data structure, the insertion, deletion, and location of various records is facilitated. Figure 15 is a logical representation of the linked list. The top linked list provides microroutine information; each link contains the name of a microroutine which serves as a key for locating a specified microroutine, a pointer to the next microroutine, and a pointer to the first microinstruction in that microroutine. The remaining links within the structure contain data for one microinstruction. This information consists of a record holding a sequential count of the microinstructions within one microroutine, the hex value of the microinstruction organized into three fields corresponding to the major hardware components on the AM29203 Evaluation Board, a set containing each class of all mutually-dependent fields, and the choice made by the microprogrammer for each class. The use of the set and the choices will be further explained in later sections. Figure 16 graphically represents the structure of a microinstruction node; Figure 17 is included to show the actual Pascal code used to create both the microroutine and the microinstruction nodes in the list. The count is used to consecutively number the microinstructions within a microroutine, and it is used as a

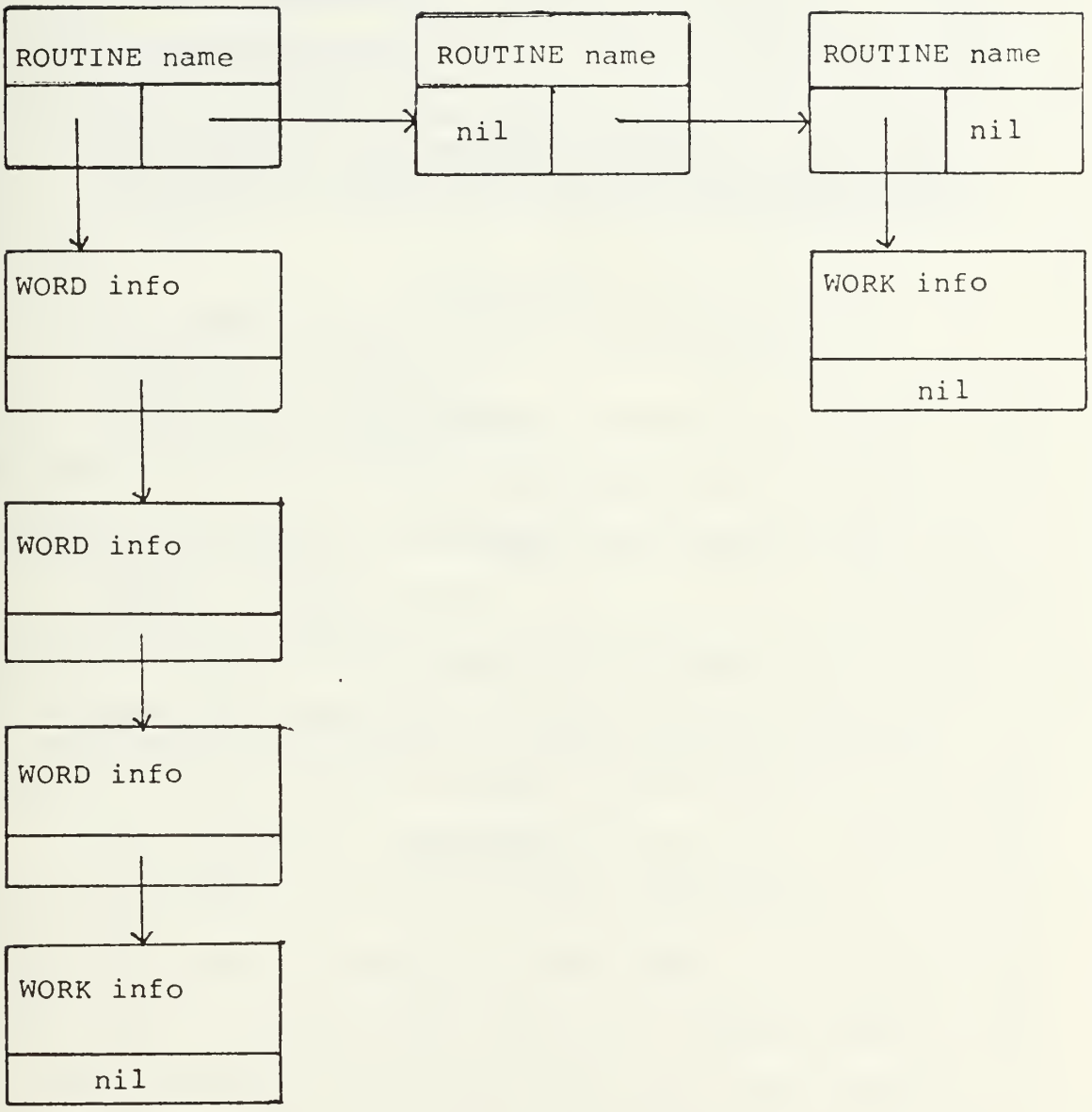

Figure 15
Logical Representation of Linked List

COUNT	AM 29203	AM 2904	AM 2910
MICRO	MACRO	CARRY	CTEST	YOUT
RAM	BRANCH	SHIFT	COMMAND	PASS
MICRO choice	MACRO choice	CARRY choice	CTEST choice
CTEST2 choice	YOUT choice	SHIFT choice	COMMAND choice

Figure 16
Microinstruction information Node

key to locate or insert a microinstruction. The last item is a pointer to the succeeding microinstruction. All links in the data structure are dynamically provided by the Pascal environment.

All of the menus have the same general format. All of the legitimate choices are shown with an alphanumeric character to indicate the response desired from the microprogrammer, the current mnemonic for the field, and an English languge summary of the mnemonic. After the choices are enumerated, a HELP option is provide which will direct the microprogrammer to various references. The last choice is a RETURN which will save the current status of the microinstruction or microroutine and return the microprogrammer to the next higher level of menus in the hierarchy. The microinstruction and microroutine menus also provide the mechanism to destroy the current microinstruction or

```
type  fieldtype = packed array [1..4] of char;
      CONFLICTtype = (micro,macro,carry,ctest,yout,
                      RAM,branch,shift,command,pass);
      nametype = packed array [1..8] of char;

      ROUTINEptr =  ROUTINErecode;
      WORDptr =  WORKrecord

      WORDinfotype = record
        count: integer;
        AM29203,AM2904,AM2910: fieldtype;
        CONFLICTclass: set of CONFLICTtype;
        MICROchoice,MACROchoice,CARRYchoice,CTESTchoice,
        CTEST2choice,YOUTchoice,SHIFTchoice,COMMANDchoice:
           char
      end;

      (* The Microroutine Node *)
      ROUTINEtype = record
        ROUTINEname: nametype;
        ROUTINEnext: ROUTINEptr;
        ROUTINEfirst: WORDptr
      end;

      (* The Microinstruction Node *)
      WORDtype = record
        WORDinfo: WORDinfotype;
        WORDnext: WORDptr
      end;

var   ROUTINElist: ROUTINEptr (* first node in list *)
      ROUTINEtop: ROUTINEptr (* current microroutine *)
```

Figure 17
Declaration of Master Data Structure

microroutine and adjust the pointers within the linked list. An example of a typical menu is found as Figure 18.

```
            MODIFY AN EXISTING MICROROUTINE MENU
What do you want to do?
   Type a  C to CHANGE the name of the Microroutine
        M to MODIFY a Microinstruction
        A to ADD a Microinstruction
        I to INSERT a Microinstruction
        D to DELETE a Microinstruction
        L to LIST a Microinstruction
        H for HELP with this menu
        R to RETURN and SAVE the current Microroutine
        A to RETURN and ABANDON the current Microroutine
```

Figure 18
Typical System Menu

D. STRUCTURE OF THE PROGRAM

The primary organization of the program is based upon the functions that the microprogrammer will perform during a terminal session. These functions are a natural hierarchy, and both the requests for menus and the structure of the PASCAL code represent this hierarchy. Figure 19 is a functional chart showing the actions that a microprogrammer would make when building a microroutine once each routine has been expressed in a pseudo-code algorithm. Figure 20 illustrates the organization of the program and how it parallels the previous functional chart.

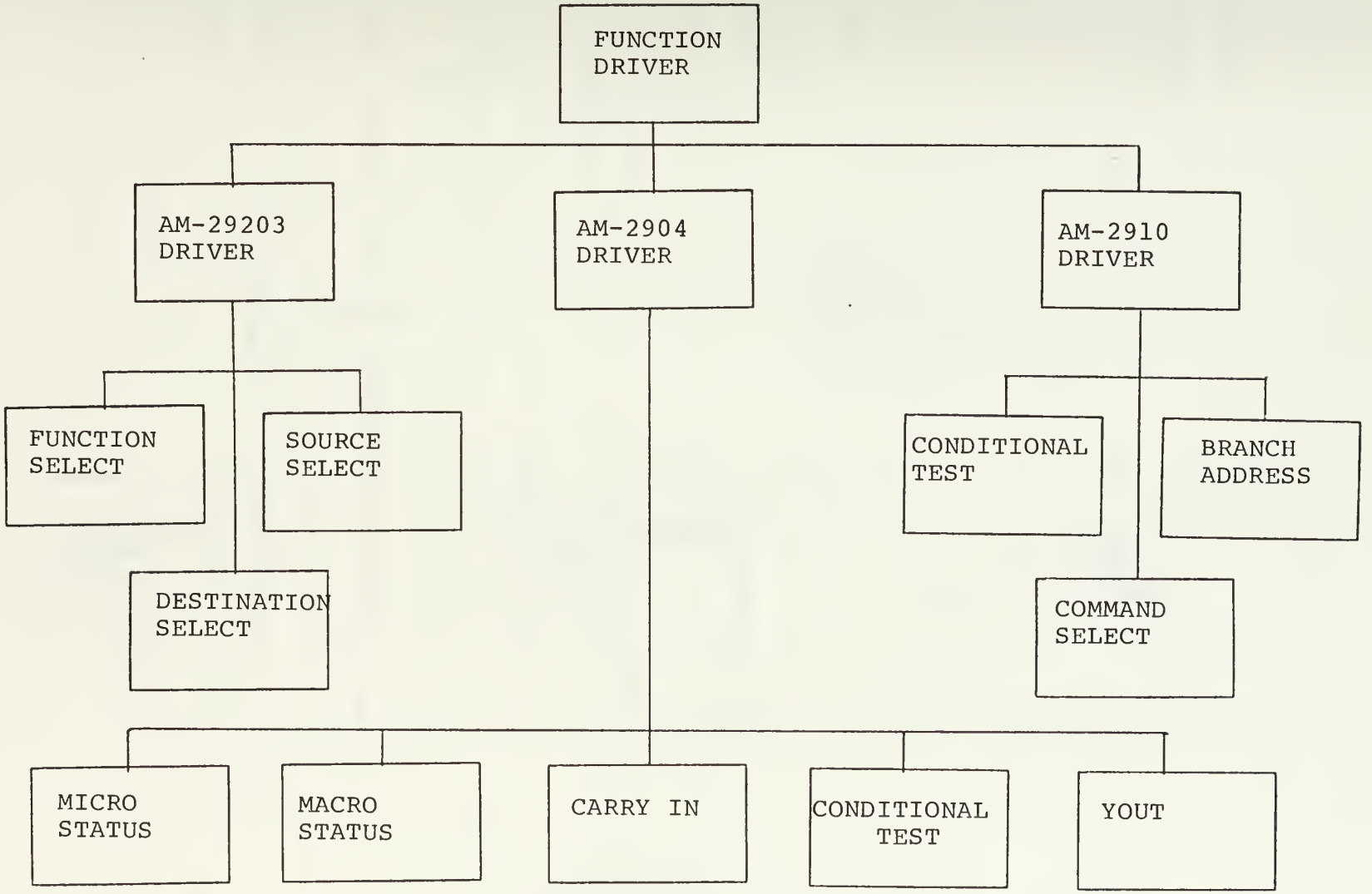

Figure 19
Function Chart

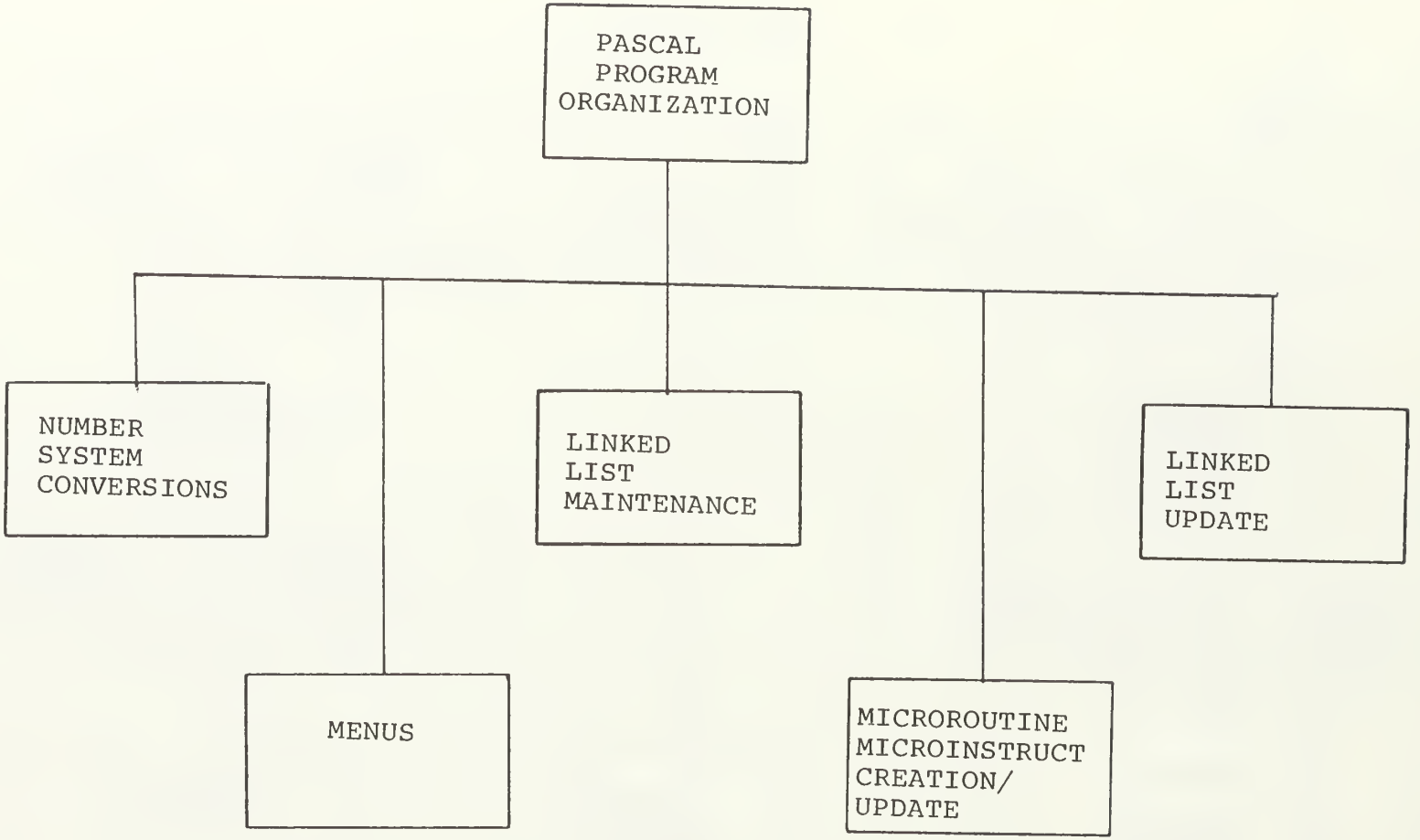

Figure 20
Program Organization Chart

The additions to the program hierarchy chart are required for manipulating the linked list and providing for conversions between hex, decimal, octal, and binary numbers. Manipulations of the linked list occur at three levels: system entry and exit, microroutine manipulation, and microinstruction actions. Upon system entry, the linked list is built based upon the contents of a disk file. This file contains all microroutines and their associated microinstructions created/modified and saved from previous terminal sessions. At system exit, all previous routines which have not been explicitly deleted and those routines which were added/modified during the session will be saved back to the same disk file. The microprogrammer also has the option to abandon all previous and current microroutines. Since the dynamic allocation of links by the system is the method used to acquire the nodes for the linked list, all of these nodes are deallocated at system exit. A microprogrammer is given the ability to scan the names of all microroutines and to receive a hardcopy report of all the microinstructions within their respective routines. The microroutine manipulation procedures which can be performed on existing microroutines include location, deletion, modification, and on-line listing of all microinstructions. A new microroutine can also be created and added to the end of the microroutine linked list.

The last level of linked list global procedures are those which provide for microinstructions. It is possible to locate, modify, and delete a microinstruction based upon the contents of the count field. This count field can be obtained by the microprogrammer by listing in the terminal the microroutine currently pointed to by the pointer ROUTINE top. A microinstruction can be inserted between two existing microinstructions based upon the count field of the preceeding microinstruction, and a new microinstruction can be added to the end of a microinstruction linked list. Each of the microinstruction procedures contains the mechanism to a;;pcate consecutive integers in the count fields for an entire microroutine.

Conversion routines are required because the final format of the microinstruction in each of the nodes of the microinstruction linked list is three fields each containing four hex numbers. While several choices from the menus are hex numbers which can be easily placed into the correct hex position within the microinstruction, some of the fields affected may be one to three bits in length. These fields will be worked on at the binary or octal level. A binary-to-hex conversion is needed to create the final format which is stored into the nodes.

Auxiliary warning menus are also provided. These menus warn the microprogrammer about the requirements for succeeding or preceeding microinstruction values which depend upon

a choice made in the current microinstruction. For example, if the microprogrammer chooses the instruction register for the operand R and operand S sources to the ALU, then she will be warned about the requirement to load the instruction register in a preceeding microinstruction. If a microinstruction field conflict exists, a warning will also be posted. Suppose that the microprogrammer created a microinstruction where both the ALU Source Field and the Sequencer command required a value to be placed into the Branch Address Field. She would receive a warning that a field conflict existed.

IV. USING THE PROPOSED MICROPROGRAMMING SYSTEM

A. DESIGN PROBLEM OF THE AM2904 SHIFT/STATUS CONTROL CHIP

The problem of mutually-dependent fields is most crucial with the I5-IØ bits for the AM2904 Shift/Status Control chip. These six bits determine the action for the micro and the Macro status registers, the carry-in to the AM29203 ALU, the register to be tested and the condition to be tested when a conditional test is performed to determine the Branch Address for the AM2910 Sequencer, and the Y output from the AM2904. It is not enough that there are five classes of actions which are controlled by these six bits, but a single choice within a particular case might be represented by one

to seven different values. If the microprogrammer desires to directly load the Macro status register, there are seven possible values that she could use. The task of writing a microinstruction based upon a detailed algorithm becomes quite difficult and confusing when these six bits are tackled.

In the meta-assembler method of microprogramming, a microprogrammer would have to remember the choices of these six bits in any of the five classes of action needed. The microprogrammer would also have to remember all the possible values for a specific choice and resolve all the conflicts by hand. As an example of the complexity of the task and the memorization required, suppose that the hypothetical microprogrammer desires to load the Macro Status Register direct and the carry-in will originate from the micro status register. A loadMSRdirect has seven possible bit patterns and the microcarry has three. Each bit in these patterns will be either a '1', a '0', or an 'X'. The 'X' represents a don't-care condition and will match a '1' or a '0'. The loadMSRdirect pattern of '1X101X' conflicts with the microcarry pattern of '0XX1XX". However, the loadMSRdirect pattern of '0X11XX' does match, and there is not a conflict among the bit patterns for the two classes of action chosen. A worst case match search would involve seven choices for both the Macro and the micro status registers, three choices for the carry-in, and one choice each for both

the conditional test and the Y output. This is a total of nineteen bit patterns that must be remembered and which are involved in trying to resolve the conflict of mutually-dependent fields.

A microprogrammer should not be required to have to remember all the patterns and resolve conflicts manually. The chance for error is greatly increased for a manual microprogramming system. It would be easy to forget to include a pattern in the search, make a mistake in comparison, or to memorize a pattern incorrectly. The microprogrammer should only need to indicate the action desired in each of the five classes, and an invisible system should find a match or report an unresolvable conflict. The proposed microprogramming system will provide this facility. A data structure is used which stores all the possible bit patterns for the choice made in each of the five classes of action. This data structure is examined each time the microprogrammer makes a choice involving these six bits; either a match is found among the classes, or the microprogrammer is informed of an unresolvable conflict.

In the program, all of possible values for bits I5-IØ of the AM2904 portion of the microinstruction are stored. For example, the bit pattern to swap the Macro status register with the micro status register, when chosen from the micro status register menu, is stored in a packed array named SWAPmsr. All seven values for a loadMSRdirect are stored in

```
type  STATUStype = packed array [1..6] of char;
      CHOICEclass = (micro .. yout);

      STATUSptr =  STATUSrecord;

      (* This record is the actual node in the linked list *)

      STATUSrecord = record
        status: STATUStype;
        next: STATUSptr;
        right: STATUSptr
      end;

      CHOICEname = packed array [1..20] of char;
      hexrange = ('1','2','3','4','5','6','7','8','9',
                  'A','B','C','E','F');

var   MICROptr,MACROptr,CARRYptr,CTESTptr,YOUTptr: STATUSptr;
      MIRCOtop,MACROtop,CARRYtop,CTESTptr,YOUTptr: STATUSptr;

      CHOICES: array [CHOICEclass] of CHOICEname;
      CHOICEset: set of CHOICEclass;

      (* The following arrays contain the actual bit patterns
         for the choice that they represent.  These values
         become the nodes in each of the linked lists.  *)

      resetSIGN, setSIGN, resetOVERFLOW, setOVERFLOW,
      loadmsr, setmsr, SWAPmsr, resetmsr, loadOVERFLOW,
      LOADMSRY, setMSR, SWAPMSR, resetMSR, invertMSR,
      carrymicroinvert, carryMACROinvert, CARRYO, CARRYl,
      CarryCx,SIGNexor, SIGNexnor: STATUStype;

      loadCARRYmsr, loadcarrymsr: array [1..2] of STATUStype;
      loadDIRECTmsr, loadDIRECTMSR: array [1..7] of STATUStype;
      microcarry, MACROcarry: array [1..3] of STATUStype;
      MICROtest, MACROtest, STATUStest:
        array [hexrange] of STATUSTYPE;
```

Figure 21
Declaration of AM2904 Data Structure

a seven-position array with each position holding the packed array of a possible bit pattern. Three data structures are used to automate the assignment of a value to bits I5-IØ. These structures are an array, a set, and a linked list. The Pascal code used to allocate the data structures is included as figure 21; it is included to assist in understanding the solution of mutually-dependent fields in the AM2904 portion of the microinstruction. The first

MICRC	load msr direct
MACRC	no choice made
CARRY	micro carry in
CTEST	micro carry
YOUT	no choice made

Figure 22
The Array Data Structure

structure is a one-dimensional array which is shown for a hypothetical case in figure 22. It is indexed by the five classes of action, and each of the five positions are initialized to 'no choice made'. A description of the action chosen by the microprogrammer for each class is stored in this array. As an illustration of the example array, the microprogrammer desires to load the microstatus register direct. The position in the array indexed by micro would contain the character string "load msr direct."

The second structure used is a set consisting of all the five classes of action. The names of the classes are the same as the index of the above described array. Only one choice per class is allowed. The set is used to enforce this rule. Each time the microprogrammer makes a choice which affects the value of bits I5-IØ, the class of that choice is determined by the program and that class's status in the set is checked. If the class is in the set, the microprogrammer has already made a choice in that class for the current microinstruction. The old choice will be removed from the third data structure and replaced by the new decision. If the class is not in the set, then this is the first time that a choice in that class has been made; the set will reflect the current status of the class, and the new choice will be added to the third data structure. If the hypothetical microprogrammer has chosen to activate the Macro status register and perform a carry-in, macro and carry will be in the set; micro, ctest, and yout will not be in the set.

The last and most important data structure is a linked list which is walked either to find a match or to determine if an unresolvable conflict exists. The format for each of the nodes and the hypothetical example completed as a linked list are shown in figure 23. When the microprogrammer makes a choice involving bits I5-IØ, a vertical linked list is

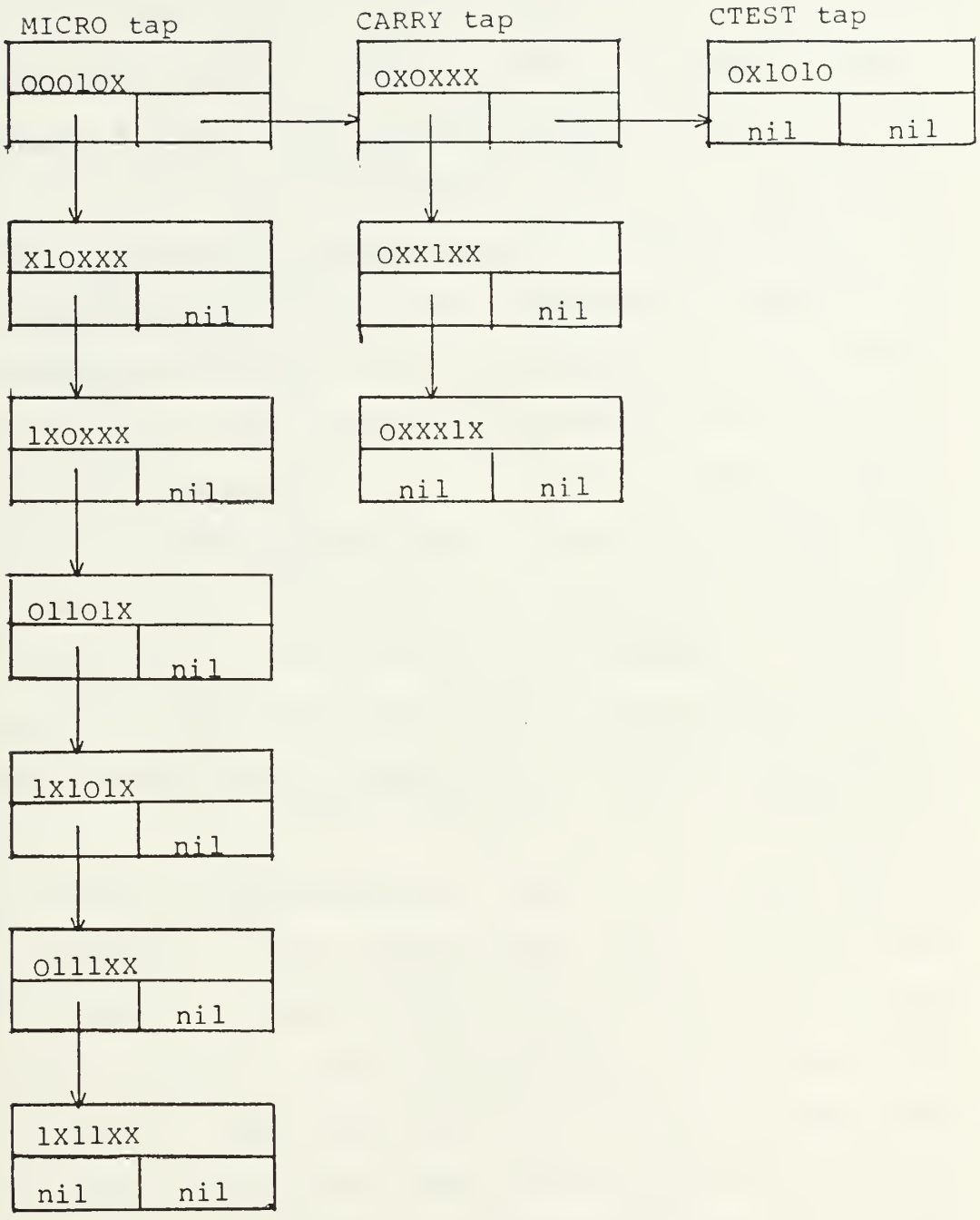

Figure 23
Logical Representation of AM2904 Linked List

built to hold all the possible bit patterns for that choice. The number of links for a choice may range from one to seven. Each class of action chosen by the microprogrammer for the current microinstruction will have its own vertical linked list. If the microprogrammer has decided to check the status of both the Macro and the micro status registers and perform a conditional test, there will be three vertical linked lists. The first two will contain seven nodes, and the last list will contain only one node. Each time that a choice is made, a vertical list is built and the entire structure is searched to check for conflicts and determine the value to be placed into bits I5-IØ. If a class of action is chosen a second time for the current microinstruction, the old vertical linked list must first be removed and replaced by the list representing the most recent choice in that class.

The search for a match involves comparing nodes in each vertical list until a node in each list is compatible with a node in every other list. At the start of a search, the first node of the first vertical list is compared against each node in order in the second list until a match is found. If a match is not found and there are more nodes in the first list, the process is repeated with the second node of the first list and all nodes in order of the second list. This iterative process continues until a match is found or no nodes remain in the first vertical list. Once a match is

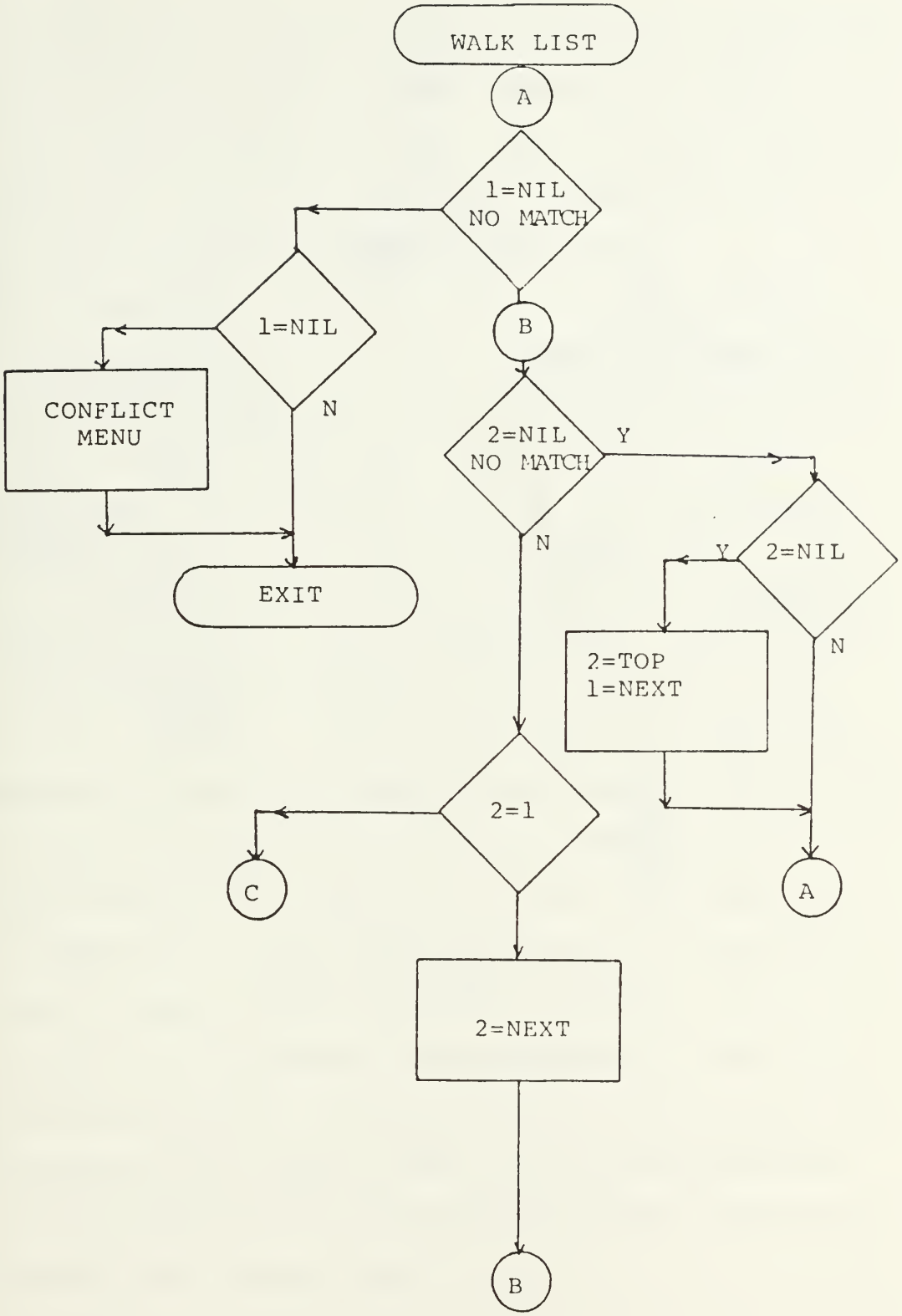

Figure 24
Logic to Walk AM2904 Linked List

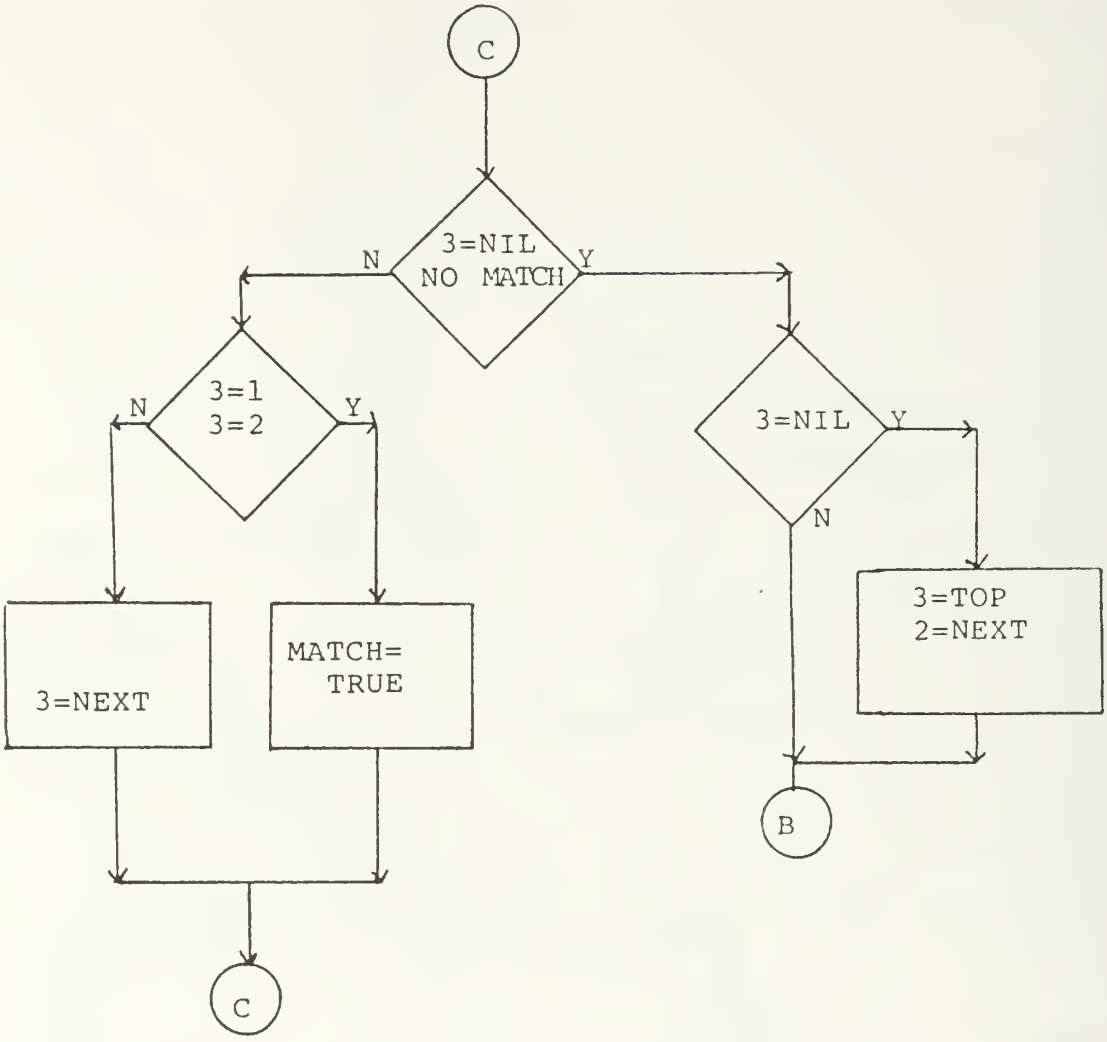

Figure 24 Con't

found between the first and second lists, links in the third list are compared with the current choice in the second list. If a match is found between these two lists, then the bit pattern from the third list must be compared to the current node in the first linked list.

This process will continue with all remaining lists until a match is found or all the choices among the various classes have been exhausted. The indication of an unresolvable conflict is when the first vertical list no longer contains nodes to be used as the base for the search and a match has not been found. The algorithm used to walk the third data structure is included in flowchart form as Figure 24. This example only processes three lists. More lists could be processed by a further nesting of the code. Since the horizontal ordering of the vertical lists is random, the lists cannot be pointed to by MICROptr, MACROptr. etc.. They are referred to in the order in which they appear as FIRST, SECOND, etc., and a permanent pointer to the top of each list is used and named topFIRST, topSECOND, etc.. The final action of the procedure WALKCHOICES is a posting to the microinstruction of the correct value of bits I5-IØ, or the presentation of a warning menu to the microprogrammer. This warning menu will show the microprogrammer the choice that she has made in each class.

B. UPPER LEVEL MENUS

There are four upper level menus which the micropro-gramer must use before she can begin work on the separate

AM2900 Family Microprogramming System
Master Menu

What do you wish to do?

Type a B to BUILD a new Microroutine
M to MODIFY an existing Microroutine
D to DELETE an existing Microroutine
S to SCAN the names of existing Microroutines
L to LIST a specific Microroutine
P to PRINT a hardcopy listing of all Microroutines
H to HELP with this menu
R to SAVE all work and RETURN to system level
A to ABANDON all microroutines and RETURN

Figure 25
Master Menu

fields in a microinstruction. The first menu is shown as Figure 25. The menu is the Master Menu to the system, and it presents to the user the ability to perform all desired microprogramming functions. As functions are completed, con-trol will return to this menu until the user indicates that she is finished. The choices available are build a micro-routine, modify an existing microroutine, scan the list of microroutine names, list a specific microroutine, delete a microroutine, and print a hardcopy report of all microrou-tines. A help option is provided. The last two choices provide for system exit; either all microroutines are saved to a disk file or all microroutines are destroyed.

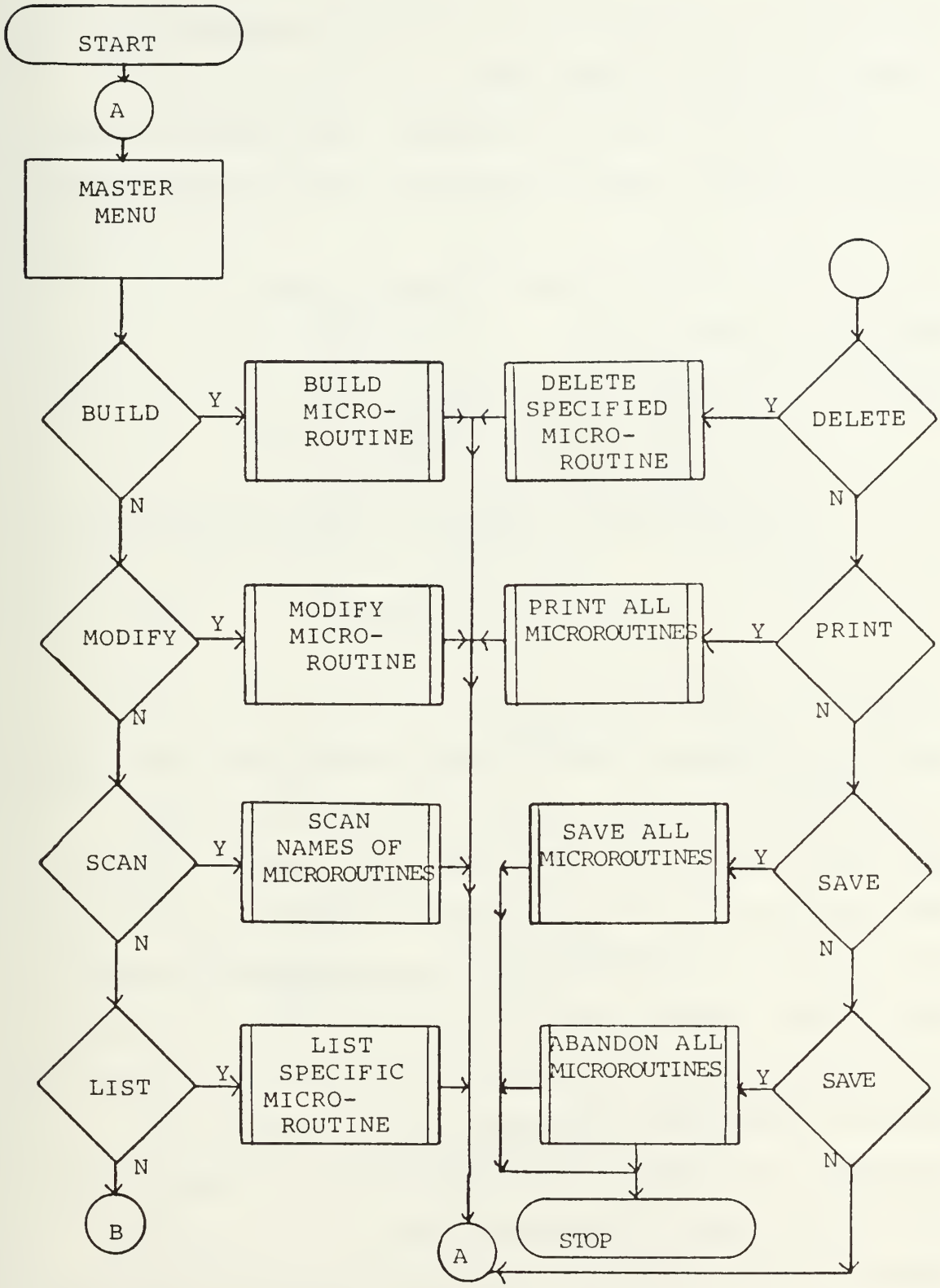

Figure 26
Overview of System

The logic for this menu and a flow of control for the entire system is shown in Figure 26.

Only two choices from the Master Menu generate other series of menus. These choices are build Build New Micro-

Build a New Microroutine Menu

What do you want to do?

Type a N to NAME a Microroutine
W to BUILD the new Microinstruction
L to LIST the Microroutine
H for HELP with this Microroutine
R to RETURN and SAVE the current Microroutine
A to RETURN and ABANDON the current Microroutine

Figure 27
Build New Microroutine

routine and Modify Existing Microroutine. The menu for build a new microroutine is found as Figure 27. Upon entry to the procedure which processes this menu and calls subordinate procedures associated with specific choices, a new microroutine node is created and added to the end of the microroutine linked list. The microprogrammer can name the microroutine, build a microinstruction, list the microroutine, receive help, and return to the Master Menu either by saving or destroying the microroutine. The user is prohibited from adding a microinstruction or listing the microroutine until it has been named. If the new microroutine is to be destroyed prior to returning to the Master Menu, the microroutine and its associated microinstructions

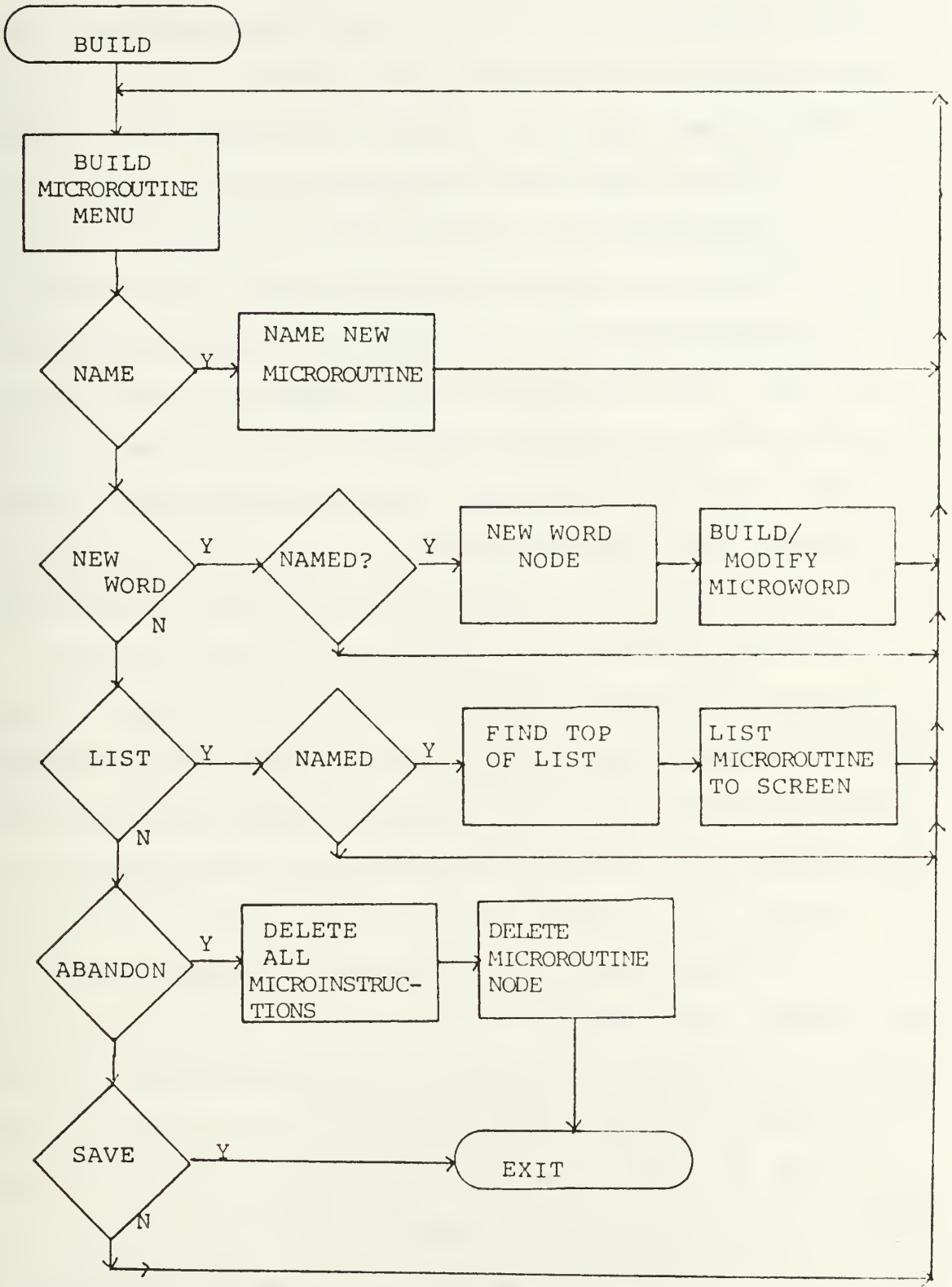

Figure 28
Logic for Build New Microroutine

will be correctly adjusted. The logic used to build a new microroutine is illustrated in Figure 28.

The actions which can be taken in modifying an existing microroutine are based upon the actions permissible in building a microroutine. Once the microprogrammer has provided the name of the microroutine to be modified, a pointer named ROUTINEtop will point to the correct microroutine link. The microprogrammer can change the name of a routine, modify an existing microinstruction, list the entire microroutine, and add a new microinstruction. This last choice, add a microinstruction, causes a new microinstruction node to be added to the end of the current vertical list pointed to by ROUTINEtop. Two additional capabilities with this menu are insert and delete a microinstruction. All of the choices are shown in Figure 29 which presents the Modify Microroutine Menu. An insert of a microinstruction creates a new microinstruction node, but

Modify an Existing Microroutine Menu

What do you want to do?

Type a C to CHANGE the name of the Microroutine
M to MODIFY a Microinstruction
A to ADD a Microinstruction
I to INSERT a Microinstruction
D to DELETE a Microinstruction
L to LIST the current Microroutine
H for HELP with this menu
R to RETURN and SAVE the current Microroutine
A to RETURN and ABANDON the current Microroutine

Figure 29
Modify Microroutine

this node is inserted before the microinstruction specified by the microprogrammer. The count field is used as a key to find the succeeding microinstruction. The count field is also used as a key to find the microinstruction to be deleted. The system will match the count provided by the microprogrammer with the count field for either the succeeding microinstruction for an insert or with the correct microinstruction for a delete. For both the insert and the delete options, the pointers within the vertical linked list must be adjusted and the count fields recomputed to ensure a sequence. The flow of control for microroutine modification is shown in Figure 30.

The last menu and its controlling procedure build or modify the various fields of a microinstruction. Whenever a microprogrammer chooses to add or insert a microinstruction, a new microinstruction node is built and correctly placed into the vertical linked list pointed to by ROUTINEtop. The Build/Modify Microinstruction procedure is then entered. If the microprogrammer chooses to modify an existing microinstruction, the correct microinstruction is found based upon the count field provided by the microprogrammer. When a new microinstruction is being built, a pointer will point to the new microinstruction node which was added to the linked list. This pointer is passed to the Build/Modify procedure. The menu for this procedure is provided as Figure 31. The microprogrammer is presented with the current

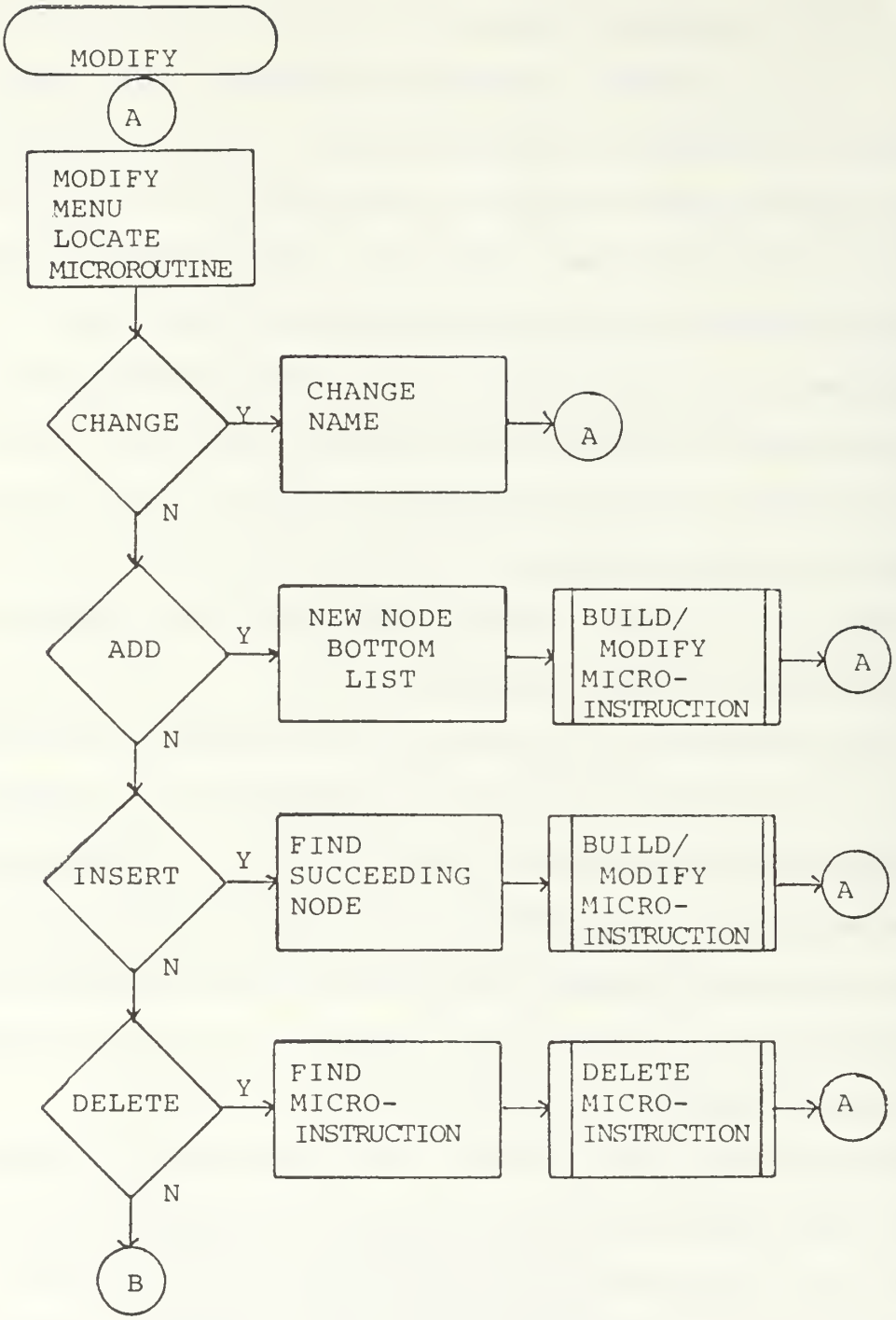

Figure 30
Modify Microroutine Logic

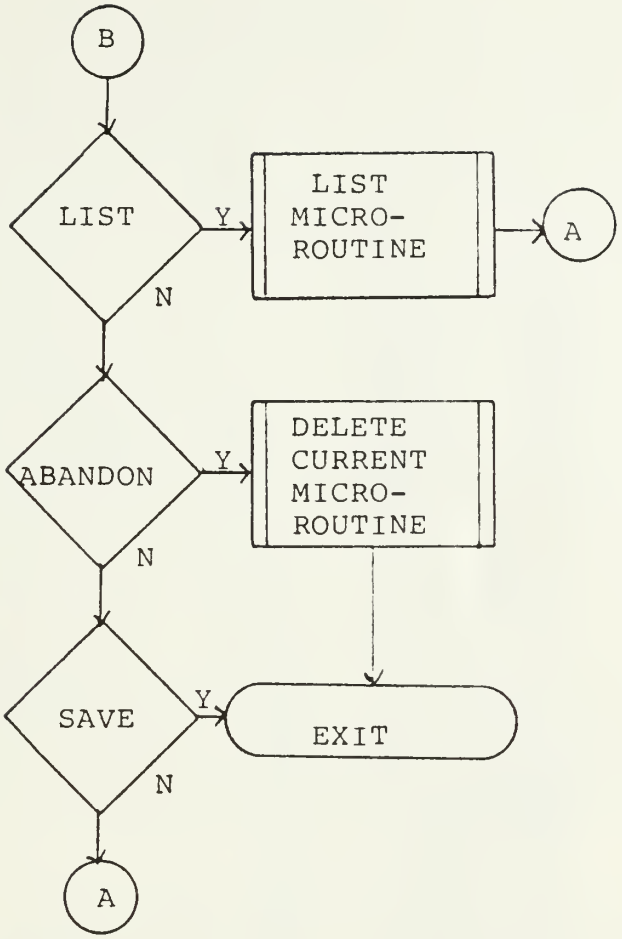

Figure 30 Con't

BUILD/MODIFY MICROINSTURCTION MENU

The current microinstruction is

Count	ALU	SHIFT/STATUS	SEQUENCER
XX	XXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXX

What do you want to do?

Type an
- A to build/modify ALU portion of Microinstruction
- S to build/modify SEQUENCER portion of Microinstruction
- M for MEMORY and Miscellaneous Commands
- H for HELP with this menu
- R to RETURN and SAVE the current Microinstruction
- A to RETURN and ABANDON the current Microinstruction

Figure 31
Build/Modify Microinstruction

binary and hex representations of the microinstruction. A default value is used for new microinstructions. The microprogrammer may choose to build/modify the ALU portion of the microinstruction, build/modify the Sequencer portion of the microinstruction, or perform miscellaneous functions such as memory writes, loading the Instruction Register, and placing an output from the AM2904 onto the Y bus. The microprogrammer can also receive help with the menu, or she can return to either the Build Microroutine Menu or the Modify Microroutine Menu. Prior to exiting this procedure, the microprogrammer must decide if the completed microinstruction is to be saved or destroyed. The logic and the recursion of this procedure is demonstrated in Figure 32.

C. THE AM29203 ALU MENUS

The most complicated part of the system is developing the ALU portion of the microinstruction. If the microprogrammer were using a manual microprogramming system, she would have to keep track of restricted ALU source choices, the need to perform up or down shifts, to make status decisions, the possible values for the carry-in, and the functions or sources which require a value in the Branch Address Field of AM2910 Sequencer portion of the microinstruction. The master AM29203 ALU procedure ensures that all fields of concern within the microinstruction are completed, provides the correct menus when choices are

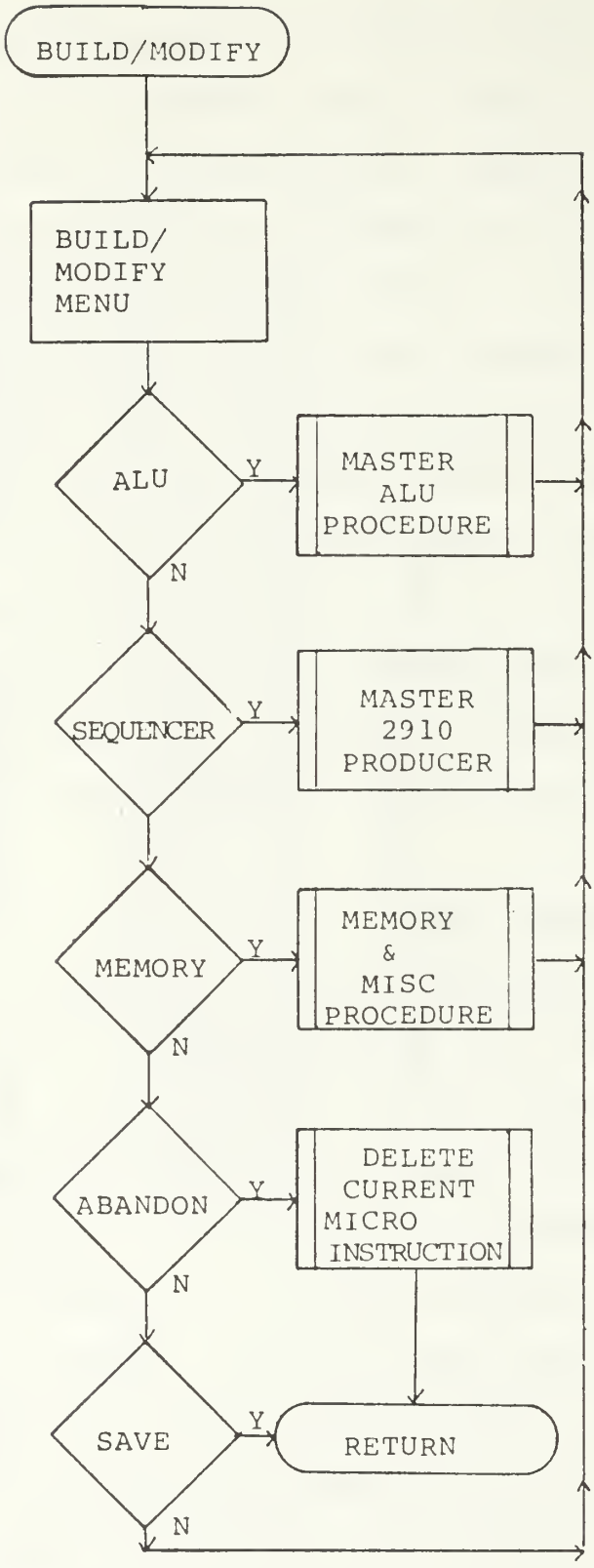

Figure 32
Build/Modify Microinstruction Logic

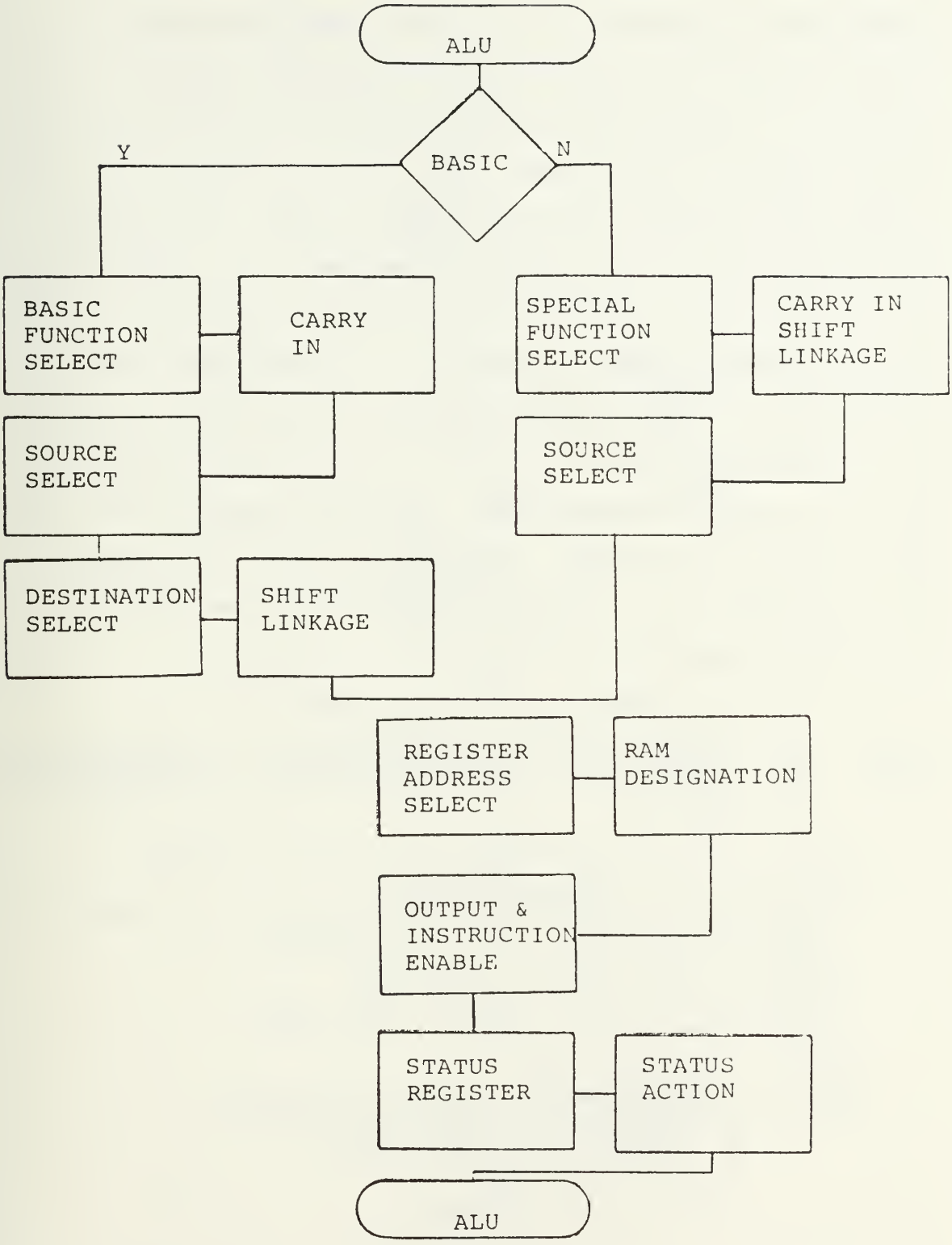

Figure 33
ALU Logic

restricted, and checks for possible conflicts among the various fields. As an aid in following the sequence of actions for completing the ALU portion of the microinstruction, Figure 33 is provided. The menus are presented in an order that provides for subordinate fields. For example, the function chosen by the microprogrammer will determine the allowable ALU operand source. The specific menu which presents the allowable choices will be displayed; a general-purpose source menu requiring the microprogrammer to remember the restrictions is not used. Field conflicts can exist with the Branch Address Field, the Shift/Command Field, and bits I5-IØ of the AM2904. The system either warns the microprogrammer of a conflict or automates the the resolution of the conflict.

```
                    MASTER AM29203 ALU MENU

Count          ALU              SHIFT/STATUS          SEQUENCER
 XX    XXXXXXXXXXXXXXXX   XXXXXXXXXXXXXXXX   XXXXXXXXXXXXXXXX
             FFFF               FFFF               FFFF

    The X's indicate bits which are not yet defined

    The defaults for the AM29203 are
      Register Address Select - bits 47-45  A,B Pipeline = 11
      Instruction Enable - bit 44 - Disable = 1                1
      Output Enable - bit 43 - Disable = 1
      Source - bits 42-40 - DAQ = 111
      Destination - bits 39-36 - YBUS = 1111
      ALU Function - bits 35-32 - OR = 1111

    What do you want to do?
    Type a  B  to choose ALU BASIC Functions
            S  to choose ALU SPECIAL Functions
            H  for HELP with this menu
            R  to RETURN to higher level
```

Figure 34

The first menu presented to the microprogrammer is the Master AM29203 Menu - Figure 34. The microprogrammer must

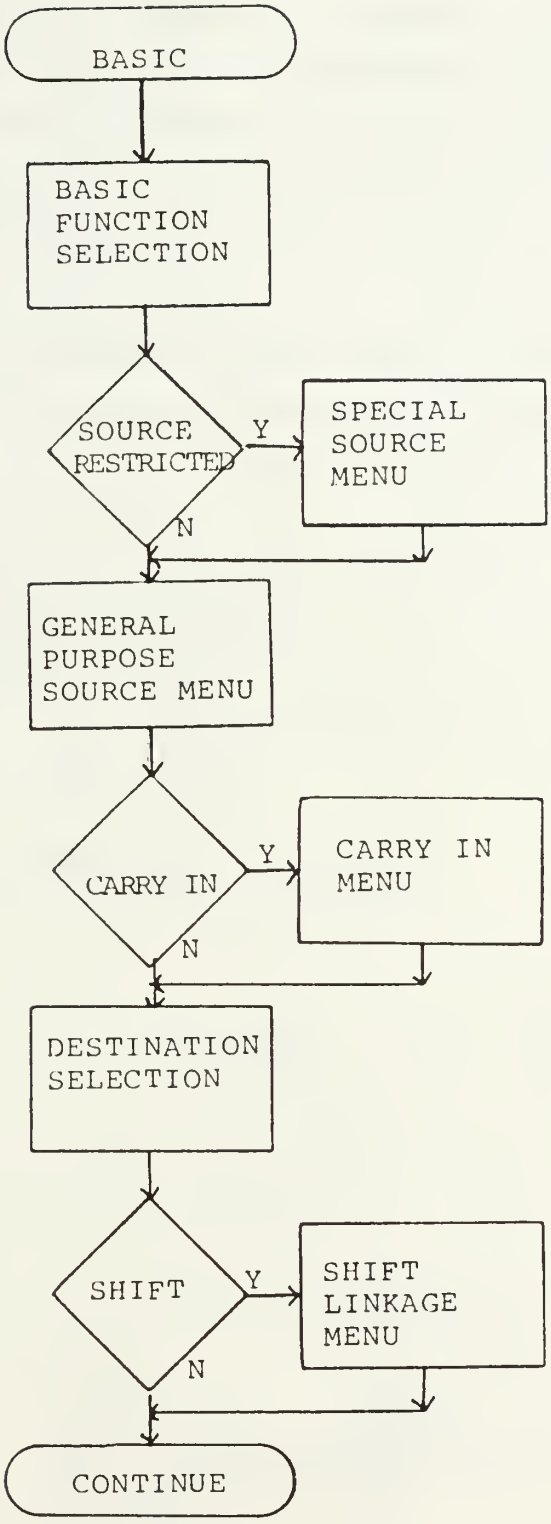

Figure 35
Flow for Basic Functions

choose a basic ALU function or a special ALU function if she desires to continue processing. Two separate procedures exist for the different choices of function. The two choices are separate in the ALU sources allowed and the destination field. The need for a carry-in or a shift linkage is also determined by different fields depending on the choice of type of function. The special functions have a restricted set of ALU sources, no destination choice, and the shift linkage is determined by the particlular function chosen. The basic functions determine either a full or restricted set of ALU source choices (whose restrictions differ from those for the special functions), require a destination choice, and base the shift linkages on the destination choice. Both functions require register address selection, output and instruction enables, and status processing.

AM29203 ALU BASIC FUNCTION MENU

Enter the value corresponding to the function you wish to perform

```
0  F = High
1  F = S - R - 1 + Carry-in
2  F = R - S - 1 + Carry-in
3  F = R - S - 1 + Carry-in
4  F = S + Carry-in
5  F = (Not S) + Carry-in
6  F = R + Carry-in
7  F = (Not R) + Carry-in
8  F = LOW
9  F = R exclusive nor S
A  F = R exclusive or S
B  F = R exclusive or S
C  F = R nor S
D  F = R nor S
E  F = R nand S
F  F = R or S
H  for HELP with this menu
R  to Return to higher level
```

Figure 36
Basic Function Selection

The Basic Function branch for microinstruction completion is shown as Figure 35. The first menu presented to the microprogrammer is the Basic Functions Selection Menu. The possible basic functions are listed in Figure 36. Should the microprogrammer choose 1 - F = High, 5 F = (not S) + Carry-In, 6 F = R + Carry-In, or 8 F = Low, the allowable

```
                AM29203 ALU SOURCE SELECT MENU

You have chosen one of the following AM29203 ALU Functions
  F = High
  F = R + Carry-in
  F = (Not R) + Carry-in
  F = Low

The only Allowed AM29203 ALU Sources are
  Operand R      Operand S      Mnemonic
  RAM A          Q Register     RAMAQ
  Direct A       Q Register     DAQ

Type a  2  for RAMAQ
        6  for DAQ
        H  for HELP with this menu
        R  to RETURN to higher level
```

Figure 37
Restricted ALU Source Selection

All other ALU basic functions allow one of the six ALU operand sources displayed in Figure 38 to be chosen. ALU basic

```
                AM29203 ALU SOURCE SELECT MENU

The Source control default is DAQ

What do you want to do?
           Operand R      Operand S      Mnemonic

Enter a 0  RAM A          RAM B          RAMAB
        1  RAM A          Direct B       RAMADB
        2  RAM A          Q Register     RAMAQ
        4  Direct A       RAM B          DARAMB
        5  Direct A       Direct B       DADB
        6  Direct A       Q Register     DAQ
        H  for HELP with this menu
        R  to RETURN to a higher level
```

Figure 38
General-Purpose ALU Source Selection

sources are restricted from a possible for six down to two. These two choices and their menu are shown in Figure 37. All other ALU basic functions allow one of the six ALU operand sources displayed in Figure 38 to be chosen. ALU basic functions 1 through 7 require a carry-in, and the Carry-In menu is included as Figure 39. Once the ALU operand

```
          AM2904 SHIFT/STATUS CONTROL CARRY IN MENU

You have chosen a function which requires a Carry-in

What do you want to do?

Type a 0 to select ZERO as the Carry-in
       1 to select ONE as the Carry-in
       2 to select Cx, the Z output of the ALU
       3 to select the carry bit from the micro status register
       4 to select the micro carry bit complemented
       5 to select the carry bit from the MACRO Status Register
       6 to select the MACRO carry bit complemented
       H for HELP with this menu
       R to RETURN to higher level
```

Carry-In Menu
Figure 39

sources and the carry-in have been chosen, the microprogrammer is presented the ALU Destination Menu which is provided as Figure 40. The Choices made from this menu determine the requirement for a shift linkage. Choices 0 through 3 and 5 require a down shift to be chosen by the microprogrammer; she is presented the menu of Figure 41. The up shift menu is presented to the microprogrammer when she chooses 8 through B or D from the destination menu. The second shift menu is provided as Figure 42. The format of these menus represents the shift linkage table that the microprogrammer would usually refer to in a manual system. [Ref. 13: p 5-18]

```
                AM29203 ALU DESTINATION MENU

Enter the value corresponding to the destination you desire

  0  RAMDA      F to RAM, Arithmetic Down Shift
  1  RAMDL      F to RAM, Logical Down Shift
  2  RAMQDA     Double Precision Arithmetic Down Shift
  3  RAMQDL     Double Precision Logical Down Shift
  4  RAM        F to RAM with PARITY
  5  AD         F to Y; Down shift Q
  6  LOADQ      F to Q with PARITY
  7  RAMQ       F to RAM and Q with PARITY
  8  RAMUPA     F to RAM, Arithmetic Up Shift
  9  RAMUPL     F to RAM, Logical Up Shift
  A  RAMQUPA    Double Precision Arithmetic Up Shift
  B  RAMQUPL    Double Precision Logical Up Shift
  C  YBUS       F to Y ONLY
  D  QUP        F to Y; Up shift Q
  E  SIGNEXT    SIGO to Y(i)
  F  RAMEXT     F to Y; Sign extend least significant bit
  H  for HELP with this menu
  R  to RETURN to higher level
```

Figure 40
ALU Destination Selection

AM2904 SHIFT CONTROL LINKAGE - DOWN SHIFT

Enter the value corresponding to the shift linkage you desire

```
0     0   -> RAMn,     0 -> Qn
1     1   -> RAMn,     1 -> On
2     0   -> RAMn,  RAMO -> Mc,     Mn   -> On
3     1   -> RAMn,  RAMO -> On
4    Mc   -> RAMn,  RAMO -> On
5    Mn   -> RAMn,  RAMO -> Qn
6     0   -> RAMn,  RAMO -> Qn
7     0   -> RAMn,  RAMO -> Qn,     Q0   -> Mc
8   RAMO  -> RAMn,   Q0  -> On,    RAMO  -> Mc
9    Mc   -> RAMn,   Q0  -> Qn,    RAMO  -> MC
A   RAMO  -> RAMn,   Q0  -> On
B    Ic   -> RAMn,  RAMO -> Qn
C    Mc   -> RAMn,  RAMO -> On,     Q0   -> Mc
D    Q0   -> RAMn,  RAMO -> kOn,    Q0   -> Mc
E    In exor IOvr -> RAMn,         RAMO  -> Qn
F    QO   -> RAMn,  RAMO -> On
H  for HELP with this menu
R  to RETURN to higher level
```

Figure 41
Down Shift Choices

```
                AM2904 SHIFT CONTROL LINKAGE - UP SHIFT

Enter the value corresponding to the shift linkage you desire

  0      0    -> RAMO,      0 -> Q0, RAMn -> Mc
  1      1    -> RAMO,      1 -> Q0, RAMn -> Mc
  2      0    -> RAMO,      0 -> QO
  3      1    -> RAMO,      1 -> QO
  4     Qn    -> RAMO,      0 -> QO, RAMn -> Mc
  5     Qn    -> RAMO,      1 -> QO, RAMn -> Mc
  6     Qn    -> RAMO,      0 -> QO
  7     Qn    -> RAMO,      1 -> QO
  8   RAMn    -> RAMO,    On  -> QO, RAMn -> Mc
  9     Mc    -> RAMO,    Qn  -> QO, RAMn -> MC
  A   RAMn    -> RAMO,    On  -> QO
  B     Mc    -> RAMO,      0 -> QO
  C     On    -> RAMO,     Mc -> QO, RAMn -> Mc
  D     On    -> RAMO,  RAMn -> QO, RAMn -> Mc
  E     Qn    -> RAMO,     Mc -> QO
  F     Qn    -> RAMO,  RAMn -> QO
  H  for HELP with this menu
  R  to RETURN to higher level
```

Figure 42
Up Shift Choices

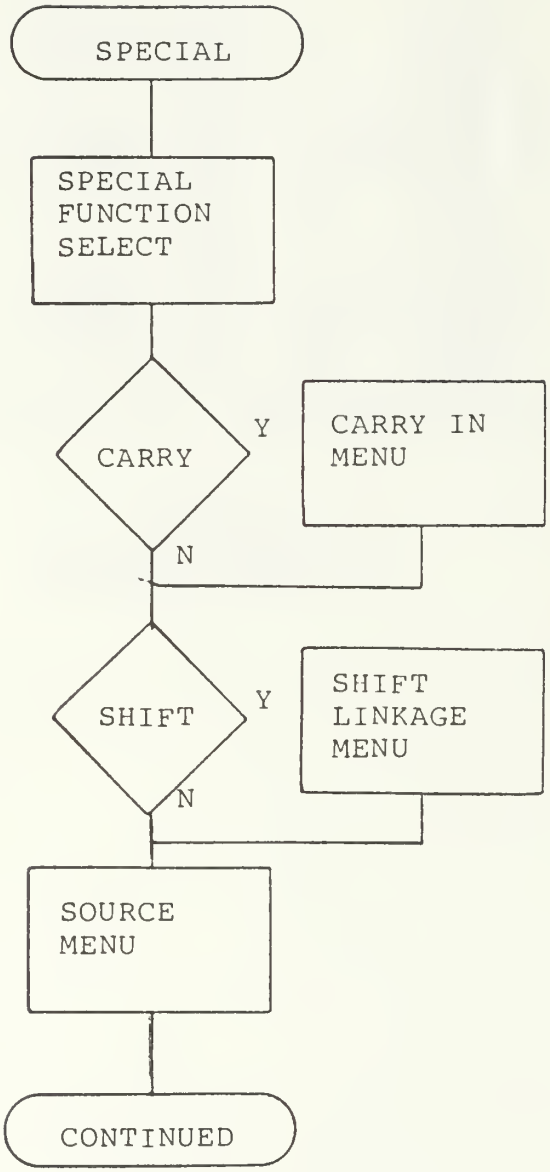

Figure 43
Processing of a Special ALU Function

AM29203 ALU SPECIAL FUNCTION SELECT MENU

Enter the value corresponding to the function you wish to perform

```
  0  Unsigned multiply
  1  BCD to Binary conversion
  M  Multiprecision BCD to Binary conversion
  2  Two's complement multiply
  3  Decrement by one or two
  4  Increment by one or two
  5  Sign/Magnitude to two's complement conversion
  6  Two's complement multiply
  7  BCD divide by 2
  8  Single length normalize
  9  Binary to BCD conversion
  Z  Multiprecision Binary to BCD conversion
  A  Double length normalize, First division op
  B  BCD ADD
  C  Two's complement divide
  D  BCD subtract F = R - S - 1 + Carry-in
  E  Two's complement divide correction and remainder
  F  BCD subtract F = S - R - 1 + Carry-in
  H  for HELP with this menu
  R  to RETURN to higher level
```

Figure 44

The actions which take place when a special function is desired are depicted in Figure 43. The process begins with the Special Function Menu - Figure 44. The choice made by the microprogrammer from this menu will determine the requirement for a carry-in and for shift linkages. There are no destination choices for the ALU special functions. Only four ALU operand sources are permitted, and the menu for these choices is shown as Figure 45. The same carry-in

AM29203 ALU SOURCE SELECT MENU

You have chosen an AM 29203 Special Function

What do you want to do?

		Operand R	Operand S	Mnemonic
Type a	0	RAM A	RAM B	RAMAB
	1	RAM A	Direct B	RAMADB
	4	Direct A	RAM B	DARAMB
	5	Direct A	Direct B	DADB
	H	for HELP with this menu		
	R	to RETURN to higher level		

Figure 45

and shift menus used by the basic functions are used by the special functions. Special function choices 0, 2-6, 8,9, and A-F require a carry-in to be chosen; a down shift is needed for choices 0-2 and 6; special function choices of 9-A will cause the up shift menu to be presented to the microprogrammer.

After these menus have been completed for the chosen type of function, the microprogrammer is ready to decide on

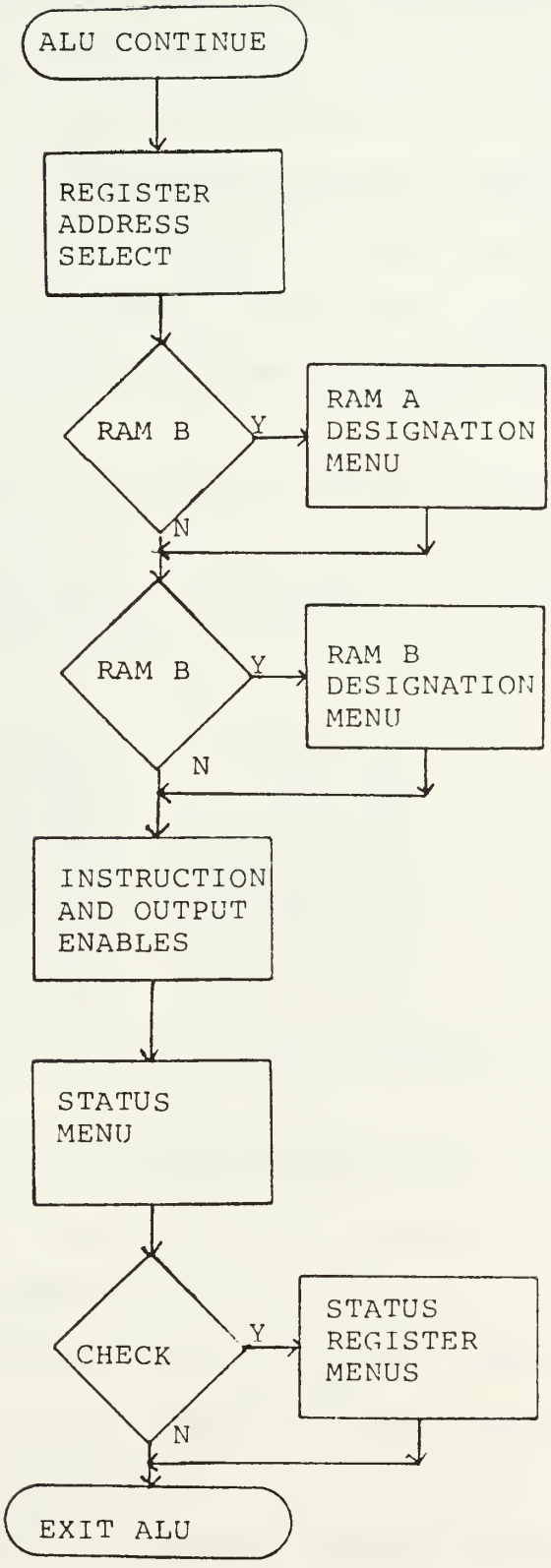

Figure 46
Remainder ALU Processing

the register address selection, the output and instruction enables, and the state of the two status registers. The flow for this part of completion of the ALU portion of the microinstruction is covered by Figure 46. This part of the overall ALU process begins with the AM29203 ALU Register Address Selection. The possible choices are shown in Figure 47. If the microprogrammer makes a choice which will cause

AM29203 ALU REGISTER ADDRESS SELECT MENU

The default is Source A - Instruction Register, Source B - Instruction Register, Destination - Instruction Register

Enter the value corresponding to the Register Address you desire

	Source A	Source B	Destination C
0	Pipeline	Pipeline	Pipeline
1	Instruction	Pipeline	Pipeline
2	Pipeline	Instruction	Pipeline
3	Instruction	Instruction	Pipeline
4	Pipeline	Pipeline	Instruction
5	Instruction	Pipeline	Instruction
6	Pipeline	Instruction	Instruction
7	Instruction	Instruction	Instruction
H	for HELP with this menu		
R	to RETURN to higher level		

Figure 47
Register Address Select

the Instruction Register to be the register address, a warning will appear telling the microprogrammer that the Instruction Register must be loaded with the correct register designation in a previous microinstruction. Should the microprogrammer choose a register address where Source A

is the pipeline, another menu will be presented which allows the microprogrammer to choose the RAM A register desired.

```
            AM29203 ALU RAM REGISTER A MENU

Enter the RAM A Register you wish to use

  0  RAM A Register 0
  1  RAM A Register 1
  2  RAM A Register 2
  3  RAM A Register 3
  4  RAM A Register 4
  5  RAM A Register 5
  6  RAM A Register 6
  7  RAM A Register 7
  8  RAM A Register 8
  9  RAM A Register 9
  A  RAM A Register A
  B  RAM A Register B
  C  RAM A Register C
  D  RAM A Register D
  E  RAM A Register E
  F  RAM A Register F
  H  for HELP with this menu
  R  to RETURN to higher level
```

Figure 48
RAM A Designation

This menu is included as Figure 48. If the Source B chosen

```
          AM29203 ALU OUTPUT AND INSTRUCTION ENABLES

Do you want the ALU results to go anywhere?

  Type a  Y  for YES;
          N  for NO


Do you want to change the contents of any ALU register
during this ALU operation-

  Type a  Y  for YES:
          N  for NO
```

Figure 49
Output and Instruction Enables

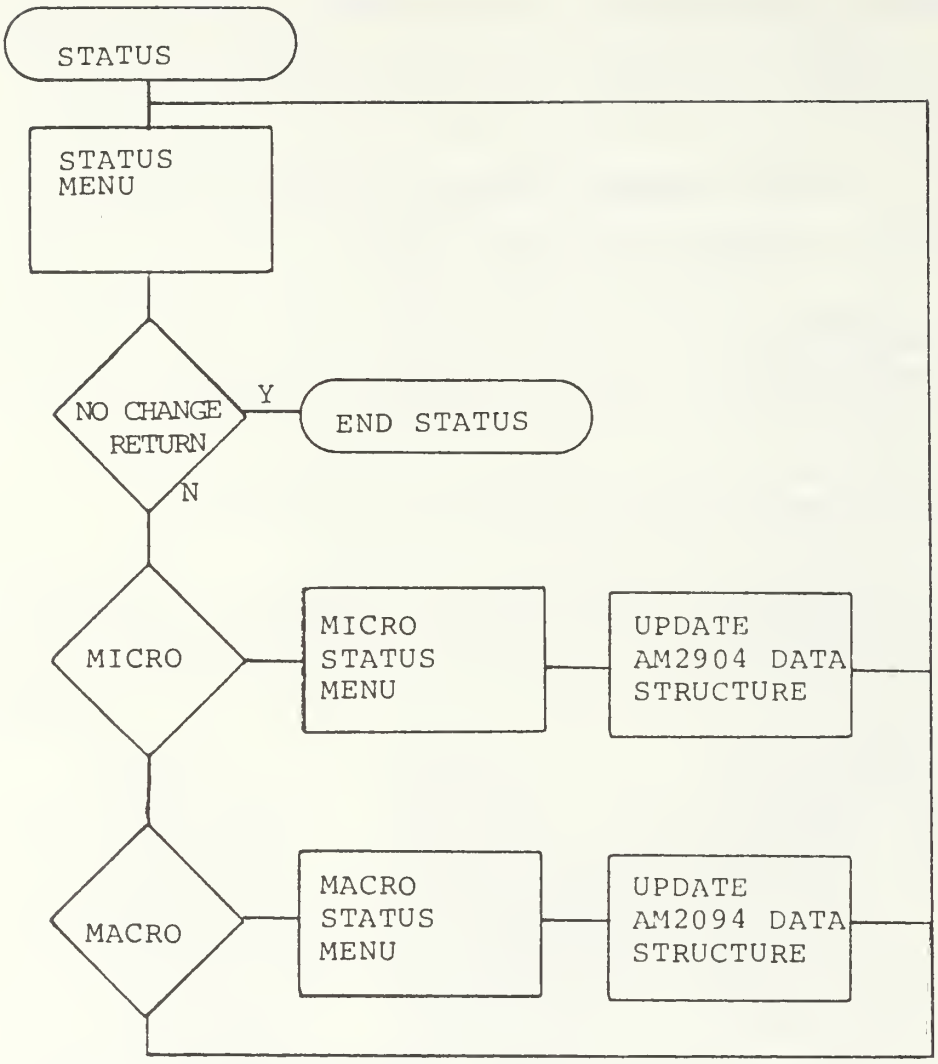

Figure 50
Status Checking

is the pipeline, a similar menu will be presented for RAM B register selection. The enable menus are very simple and require two yes or no answers. The menus for the enables are shown in Figure 49.

The last decision that must be made concerns the status registers. The logic used to implement the decision can be found as Figure 50. The bits affected are I5-IØ of the AM2904 portion of the microinstruction, and the choices from these menus interact with the data structures and the procedure WALKCHOICES described in an earlier section. The first menu which appears is figure 51; it is iteratively

```
                AM2904 STATUS REGISTER MENU

There are two status registers to control
    Micro status register
    MACRO Status Register

What do you want to do?

Type a  0  to make NO CHANGES to the stauts registers
        1  to change the Micro status register
        2  to change the MACRO Status Register
        H  for HELP with this menu
        R  to RETURN to higher level
```

Figure 51
Main Status Checking Menu

displayed until the microprogrammer indicates a choice of 0 to not change the status register or a choice to Return. Both the micro and the Macro status registers can be controlled. If the microprogrammer desires to change the micro status register, figure 52, the Micro Status Register

AM2904 MICRO STATUS REGISTER MENU

Enter the value corresponding to the action you desire

0 Load the MACRO Status Register into the Micro status register
1 Set all bits in Micro status register
2 Swap the Macro Status Register and the Micro Status register
3 Reset all bits to 0 in the Micro status register
4 Load the Micro status register from the IMMEDIATE INPUT S
5 Load all Micro status register from I, except OVERFLOW
6 Load all Micro status register from I, except CARRY bit
7 Reset only the ZERO flag in the Micro status register
8 Set only the ZERO flag in the Micro status register
9 Reset only the CARRY flag in the Micro status register
A Set only the SIGN flag in the Micro status register
B Reset only the SIGN flag in the micro status register
C Set only the SIGN flag in the Micro status register
D Reset only the OVERFLOW flag in the Micro status register
E Set only the OVERFLOW flag in the Micro status register
H for HELP with this menu
R to RETURN to higher level

Figure 52

menu will be presented. The choices from the Micro Status Register Menu will reflect in the earlier-described set named CHOICESset, the specific choice from this menu will appear in the array CHOICES, and all possible bit patterns for this choice will be entered into the I5-IØ linked list. The same actions will be taken if the microprogrammer chooses to change the Macro Status Register. The Macro Status Register Menu is included as figure 53.

AM2904 MACRO STATUS REGISTER MENU

Enter the value corresponding to the action you desire

```
0  Load the Y inputs into the MACRO Status Register
1  Set all bits if enabled
2  Swap the MACRO Status Register and Micro status register
3  Reset all bits if enabled
4  Swap the MACRO CARRY bit and the MACRO OVERFLOW bit
5  Complement all bits
6  Load all MACRO Status Register from I, Invert Carry
7  Load all MSR from I
H  for HELP with this menu
R  to RETURN to higher level
```

Figure 53

D. AM2910 SEQUENCER PORTION OF THE MICROINSTRUCTION

An equally important but less complicated portion of the microinstruction is organized around the AM 2910 Sequencer. When the microprogrammer chooses to complete this portion of the microinstruction, the Master AM2910 Menu is presented. This menu is found as figure 54. At this point, the micro-programmer will continue by indicating the desire to select

MASTER AM2910 SEQUENCER MENU

Count	ALU	SHIFT/STATUS	SEQUENCER
XX	XXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXX
	FFFF	FFFF	FFFF

The X's indicate bits which are not yet defined

The defaults for the AM2910 Sequencer are
Sequencer Command - Continue = E
Branch Address Field - 3FF

What do you want to do?

Type a 0 to select the SEQUENCER COMMAND
H for HELP with this menu
R to RETURN to higher level

Master Sequencer Menu
Figure 54

a sequencer command or return to the Build/Modify Micro-instruction Menu. If the choice is to continue, the list of sixteen sequencer commands will be presented in a menu provided as figure 55. The microprogrammer will choose one of sixteen commands.

AM2910 SEQUENCER COMMAND SELECT MENU

Enter the value corresponding to the command you desire

```
  0  JZ    Jump zero
  1  CJS   Conditional jump subroutine
  2  JMAP  Jump map
  3  CJP   Conditional jump pipeline
  4  PUSH  Push/ Conditional load register/counter
  5  JSRP  Conditional jump subroutine via register or pipeline
  6  CJV   Conditional jump vector
  7  JRP  Conditional jump via register or pipeline
  8  RPCT  Repeat loop, counter not equal 0
  9  RFCT  Repeat counter, counter not equap 0
  A  CRTN  Conditional return from subroutine
  B  CJPP  Conditional jump pipeline and pop
  C  LDCT  Load counter and continue
  D  LOOP  Test for end of loop
  E  CONT  Continue
  F  TWB  Three way branch
  H  for HELP with this menu
  R  for RETURN to higher level
```

Figure 55
Choice of Sequencer Commands

The remainder of the actions for this portion of the microinstruction is determined by the choices for the sequencer command. Figure 56 illustrates these subordinate actions. Three possible paths can be followed: No further choices are required, the Branch Address Field must be completed, and/or a conditional test is required. If no

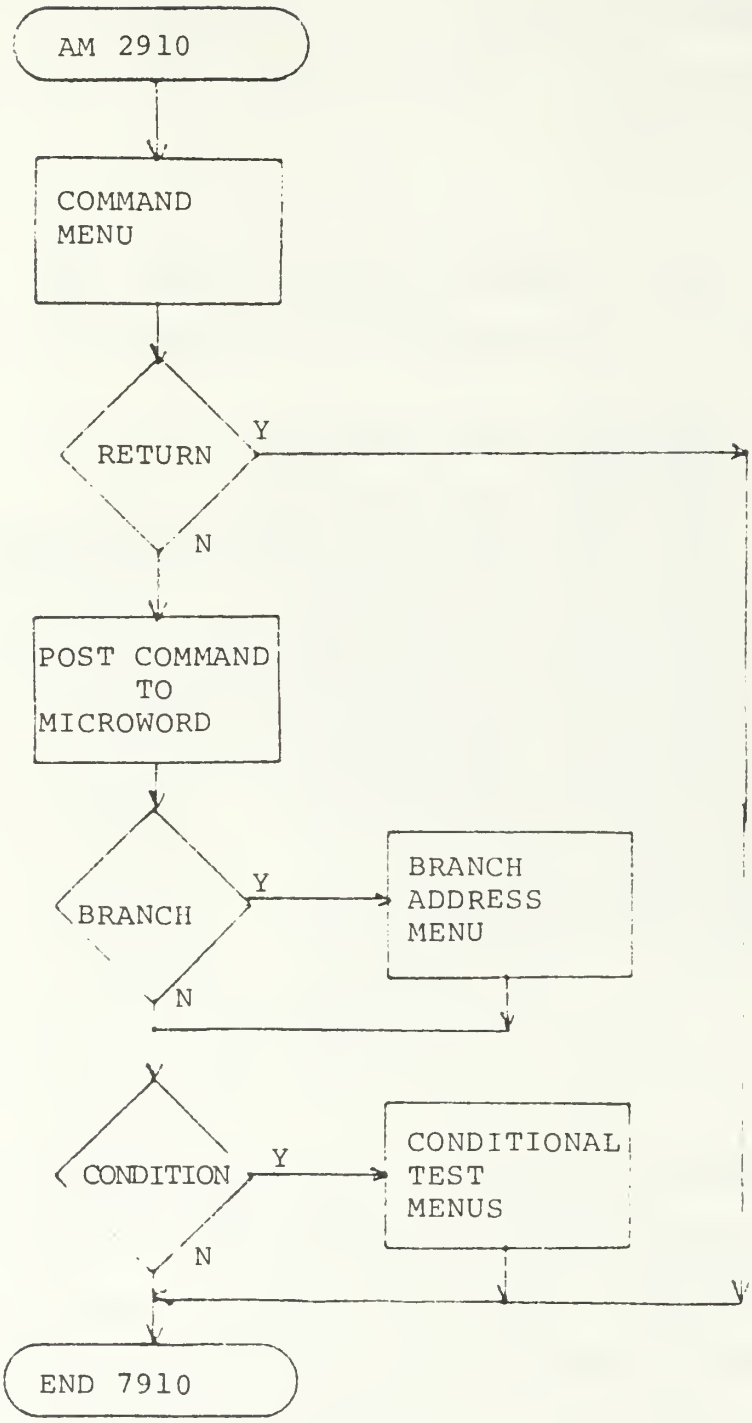

Figure 56
2910 Command Flow Chart

further action is needed, the Master AM2910 Menu will be displayed to enable a return to the upper levels of the menu hierarchy. If the selected sequencer command requires a microprogrammer supplied value for the Branch Address Field, the Branch Address Menu will be presented for completion. This menu is included as Figure 57.

```
AM2910 SEQUENCER BRANCH ADDRESS MENU

You have chosen a command which requires a value in the
Branch Address Field

The default is 3FF

Type your three-digit branch address
     a  H  for HELP with this menu
        R  to RETURN to higher level
```

Figure 57
Branch Address Field Completion

A sequencer command may provide for conditional flow of control within the microroutine. Whenever this type of command is selected, a Conditional Test Menu wil be presented. Figure 58 lists the choices available to the

```
AM2910 SEQUENCER CONDITION SELECT

You have chosen an AM2910 Sequencer Command which requires a
conditional test

What do you wnt to do?

Type a  P  for FORCED PASS
        F  for FORCED FAIL
        T  to TEST the condition
        H  for HELP with this menu
        R  to RETURN to higher level
```

Figure 58

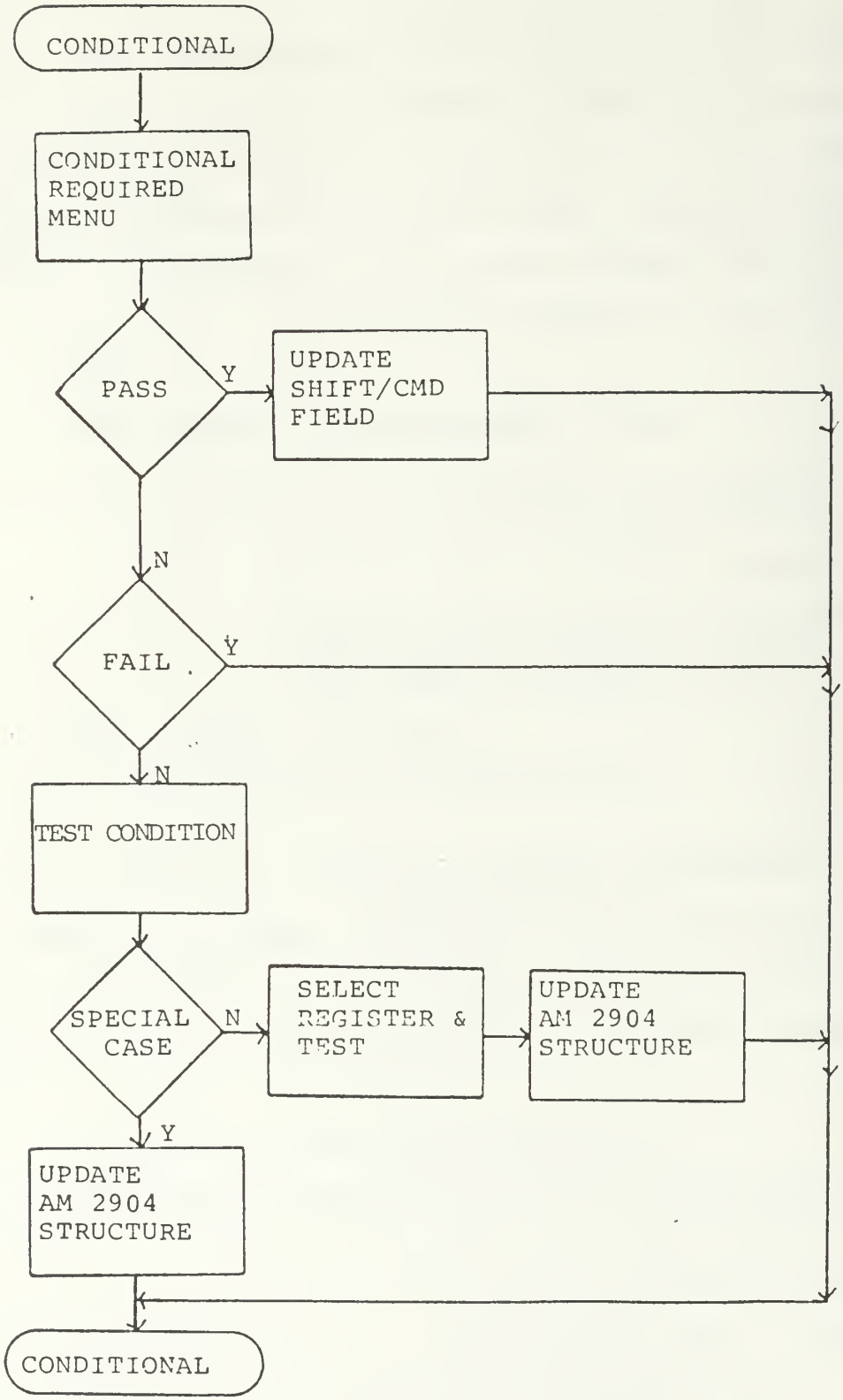

Figure 59
Conditional Test Select Flow Chart

microprogrammer; she may choose to force a pass, to force a fail, or to test a condition. If she selects to test a condition, two more menus may be required. The logic of choosing the correct condition test is included as figure 59; Figures 60 and 61 provide the two conditional test menus. First, the microprogrammer must decide what type of

```
                AM2904 CONDITIONAL TEST MENU

There are two steps to selecting a test condition
  1) select a register to be used
  2) select a test on that register

This menu selects the registers or two special tests which
combine two registers

What do you want to do?

Type a  0  for the Micro Status register
        1  for the MACRO Status Register
        2  for the IMMEDIATE status inputs
        3  for Immediate sign exor Macro sign
        4  for Immediate sign exnor Macro sign
        H  for HELP with this menu
        R  to RETURN to higher level
```

Figure 60
Conditional Register Select

test to perform. Either one of two specific tests can be done or a register for the test can be selected. If the microprogrammer chooses to test either of the two status registers or the immediate status inputs, the second conditional test menu will be presented [Figure 61]. The

```
                AM2904 CONDITIONAL TEST MENU

What condition do you want reflected by the conditional test?

Type a  0  for (SIGN exor OVR) or ZERO
        1  for (SIGN exnor OVR) and not ZERO
        2  for (SIGN exor OVR)
        3  for (SIGN exnor OVR)
        4  for ZERO
        5  for not ZERO
        6  for OVR
        7  for not OVR
        8  for (CARRY or ZERO)
        9  for (not CARRY) or (not ZERO)
        A  for CARRY
        B  for not CARRY
        C  for (not CARRY or ZERO)
        D  for (CARRY or not ZERO)
        E  for SIGN
        F  for not SIGN
        H  for HELP with this menu
        R  to RETURN to higher level
```

Figure 61
Conditional Test Choices

specific test to be performed is then chosen from this second menu. The conditional test is one of the five classes of actions that determine the bit pattern in bits 15-10. The CTEST will be set in the set CHOICEset, and the array CHOICES will reflect the test condition chosen by the microprogrammer. The bit patterns for that choice will also be added to the linked list, and this vertical list will be pointed to by the variable topCTEST.

E. MEMORY COMMANDS AND MISCELLANEOUS FUNCTIONS

If the microprogrammer requires an interface with the main memory or desires to perform some miscellaneous commands such as instruction fetch, pass a register address through the ALU into the Instruction Register, or load the Instruction Register, she will need to access the menu shown in Figure 62. This menu is called from the Build/Modify Microinstruction Menu. There are eleven possible choices, and the choice made by the microprogrammer will be reflected in the microinstruction by looking at the command enable bit and the four bits in the shift/command field. The command bit will be enabled, and the four bits will contain the choice from the menu. If the choice made by the micro-programmer is either to enable the 2904 Y output or to write the 2904 status to memory, the Y output menu is required. This menu is included as Figure 63. If the y output menu is

```
                MEMORY AND MISCELLANEOUS COMMANDS MENU

What do you want to do?

Type a  0  for OEY04       Enable 2904 Y-output
        1  for LDIR        Load Instruction Register
        2  for CONAB       Register Address thru ALU to IR
        3  for RDMEM       Read Memory
        4  for WRTMEM      Write to Memory
        5  for CONBUS      Enable constant to B-bus
        6  for IFTCH       Instruction Fetch
        A  for READ        Read enable
        B  for WRITE       Write enable
        C  for SAVESTAT    Write 2904 status to memory
        D  for DAVECON     Write constant to memory
        H  for HELP with this menu
        R  to RETURN to higher level
```

Memory and Miscellaneous Commands
Figure 62

```
                    AM2904 Y OUTPUT MENU

You can output something from the AM2904 onto the Y-bus.

What do you want to do?

Type a  0  to output the Micro status register
        1  to output the MACRO Status Register
        2  to output the IMMEDIATE inputs from the ALU
        3  for NO OUTPUT
        H  for HELP with this menu
        R  to RETURN to higher level
```

Y Output Menu
Figure 63

needed, the AM2904 design data structures will be updated since the Y output is one of the five classes of actions reflected in the array, set, and linked set. A new vertical linked list will be added containing the bit patterns for the choice from the Y output, and this new list will be pointed to by topYOUT.

The possibility of conflict in the shift/command field exists. If the ALU special function or the ALU destination chosen required a shift linkage to be established, the shift bit will be enabled and the shift linkage chosen by the microprogrammer will be stored in the shift/command field. If the bit pattern for the command action just chosen differs from the bit pattern for the previously entered shift linkage, a conflict exists. The microprogrammer must be warned. She may have to consider a different ALU special function or ALU destination, choose a compatible memory command, or perform the desired microoperations in two separate microinstructions. A shift and a memory command can only coexist in the same microinstruction when their bit patterns are identical.

A conflict may also exist whenever the microprogrammer has chosen to do a conditional test. The conditional test enable (CCEN) and the output enable conditional test (OECT) must both be a zero for the output of the conditional test to appear on the Y bus. The value in the four bits of the shift/command field which generates these zeros is a hex

"9." If the microprogrammer chooses to do both a conditional test and a memory command, a conflict will exist. None of the memory or miscellaneous commands allow for the hex value of "9." The microprogrammer will be warned if such a conflict exists as she builds the microinstruction. She will probably have to perform the desired functions with two microinstructions in the event of a conflict.

The written description of the process of creating microroutines and microinstructions is tedious and at times difficult to understand. Although samples of the menus and several flow charts are included, it seems that there are many details that must be remembered. Many actions are also occuring; not only are the fields in the microinstructions begin completed, but linked list pointers are updated and conflict checking occurs. It should be kept in mind that the microporgrammer, when using this system, is raised above the level of detail presented in this chapter. The sequencing of menus is automatic and predicated upon the user's choices. The process of checking for conflicts between mutually-dependent fields is invisible to the microprogrammer. The existence of the two linked lists, one as a master data structure and the other for the AM2904 design problem, is unknown to the microprogrammer. All that she needs to use this system is her completed algorithm and a knowledge of the hardware to be controlled. This proposed

microprogramming system provides an easy-to-use and secure method for creating microcode which solves the problem outlined in the microprogrammers' algorithm.

V. SUMMARY, QUESTIONS, AND FUTURE RESEARCH

A. SUMMARY OF MUTUALLY DEPENDENT FIELDS

The greatest contribution of the proposed microprogramming system is the handling of mutually-dependent fields. A vertically-organized microinstruction is harder to complete because several microoperations interact and use a specific field as a conflict point. A microprogrammer may desire to perform two microoperations in one microcycle. Logically, it may be reasonalbe to perform these operations at the same time, and they could be done at the same time with a horizontally-formatted microinstruction. These two microoperations may store the binary representation for the two separate actions in the same field. What happens when the binary representations are different? In a manual microprogramming system, the microprogrammer must remember that certain fields are shared and check for potential conflicts.

The proposed microprogramming system provides a mechanism for releasing the microprogrammer from the error prone and tedious process of keeping track of potential conflicts. The system will either warn the microprogrammer of a conflict, or it will attempt to resolve the conflict.

With the AM29203 Evaluation Board, three fields within the microinstruction are sites for potential conflicts: the Branch Address Field, the Shift/Command Field, and the bits I5-IØ in the AM2904 portion of the microinstruction.

The Branch Address Field is mutually-dependent upon the register address selection field associated with the AM29203 ALU and the AM2910 sequencer command field. If the register address selection indicates that the pipeline is the source for a register designation, this register designation is placed in the Branch Address Field. A sequencer command which requires a branch address or a value to be placed in the register/counter will put that address or value into the Branch Address Field. If the microprogrammer chooses a register address selection which specifies the pipeline as a source, the sequencer command will be checked. If both fields require use of the Branch Address Field, a warning menu will be displayed. Should the microprogrammer select a sequence command which causes a value or address to be placed into the Branch Address Field, the register address selection will be checked and a warning about a conflict presented if needed. It is the microprogrammer's responsibility to correct the situation.

Three other actions which interact are the requirement for a shift linkage through the AM2904, the selection of a memory command, and the need to perform a conditional test. These three actions use the four bits of the Shift/Command

field of the AM2904 portion of the microinstruction. The hex value chosen from the up or down shift menus is placed in this field as is the hex value for a desired memory command. A conditional test requires a hex "9" in this field to enable the condition codes and the enable the output of a conditional test. The shift values are in the range of "0" through "F"; the memory command values are "0" through "6" and "A" through "F." It is impossible to do a conditional test in the same microinstruction as a memory command because there is no common hex value shared between these actions. A shift linkage and a memory command can occur together only if the hex values of the shift linkage match the bit pattern for the memory command. A shift linkage and a conditional test may only occur simultaneously when the shift linkage chosen is a "9."

A specific conditional test must also be considered when discussing the Shift/Command Field - a forced pass. A forced pass will take place either when the command enable bit is disabled or when this bit is enabled and the value in the field will allow CCEN to be high. These values are "0" through "6" and "A" through"D." The shift linkage must also match the memory command. For a forced pass, it is necessary to first check the command enable bit. If it is not enabled, the proposed system will check to see if the shift

linkage value is in the correct range. Whenever a conflict is present, the microprogrammer will receive a warning menu.

The resolution of conflict in bits 15-10 of the AM2904 Shift/Status chip has already been discussed at length. The goal with this field is to examine all possible bit patterns for the choices made, and automatically find a compatible pattern.

The master data structure is used to keep track of potential conflicts. Each source of a conflict is represented by a set named CONFLICTclasses. If a microprogrammer should chose the pipeline as a register address source, RAM will be placed in the set. If the microprogrammer should choose the pipeline as a register address source, will be placed in the set. If the microprogrammer then selects a shift linkage, the member SHIFT pipeline will be placed into the set. In the determination and resolution of conflicts, the choices for the various actions are also needed. The choice for shift and RAM will contain the hex value selected by the microprogrammer from the menu. The only fields of potential conflict which do not require the maintenance of the actual values selected from the menus are the register address selection and the AM2910 sequencer command. They will only be placed into the set CONFLICTclasses if the potential exists for conflict. In some instances, the determination of conflict depends only upon the membership in the set, such as a conditional test and a

memory command. Other times the actual values of the choices must be compared to find a conflict among mutually-dependent fields.

B. STATUS OF THE PROJECT

The proposed microprogramming system is not complete. All selection menus are finished and accessible to the microprogrammer. She has the ability to complete both the AM29203 ALU and the AM2910 portions of the microinstruction. The mechanisms are also working which allow all actions on the four upper level menus to be completed. All choices from the Master Menu have been tested. The microprogrammer can also perform the additions, deletions, insertions, and modifications associated with the Build Microroutine and Modify Microroutine menus. The overall linked list structure containing the names of the microroutines and their associated microinstruction can be stored to and built from a disk file.

The major design problem has been solved. The proposed system will process conflicts between mutually-dependent fields. Earlier sections described the mechanics of comparison. This information is stored with the microinstruction because it is necessary to know the most recent choice made in each of the ten classes of action which are a list of the fields where a conflict might occur

or originate. The largest source of conflict - bits I5-IØ - has been resolved with an automated technique to find compatible values for the five classes in question. The design decision in terms of the set membership and actual choice comparison have not been implemented. The warning menus are also not complete. The data structure for bits I5-IØ has been designed, and the Pascal code for its implementation is finished, but no testing has taken place.

C. AREAS OF QUESTION

The first decision made which requires further investigation is the use of Pascal as the language for implementation of the proposed system. It is not a language well-fitted to an interactive menu driven system; no facilities exist to clear a screen or to start a menu at the top of a screen. Feature interaction in Pascal allows only for static arrays. This restriction caused a heavy reliance on linked lists because of their dynamic capabilities.

The second area of consideration is the linked lists and the format of the nodes in the master linked list structure. Was it necessary to always have the classes of conflict and the choices within each class available at all times? The nodes in the linked list were always visible to the entire system. A less visible structure which provides for

information hiding and whose purpose is the determination and resolution of conflict must be considered as a possible improvement in the system.

A last area of consideration is how the linked list is used to resolve conflict among the five classes of action which affect bits I5-IØ. Is a linked list the best approach in terms of ability to solve the design problem? Also, is a separate structure needed to determine conflict among the shift linkages, memory commands, and conditional test? Are these mutually-dependent fields of sufficient complexity to require their own data structure?

D. FUTURE RESEARCH

The main thrust of future research should be the goal of retargetability. Future researchers need to examine methods where a microprogrammer can choose various pieces of microprogrammable hardware and configure her own microinstruction to control this microprogrammer-defined architecture. Some of the concerns will be the identification of mutually-dependent fields and the compatible values that they may contain as well as the identification of conflict. The microprogrammer will also need to be able to select from existing menus or create new menus online. A linkage will also be needed from the choices on the menu to bits in the microinstruction. A future design problem will

be the automated microcode generator for a user-defined microprogrammable architecture.

E. CONTRIBUTION OF THE PROPOSED MICROPROGRAMMING SYSTEM

The AM290203 Evaluation Board is primarily used as a teaching tool in microprogramming. The architectural design considerations, for both the chip layout and the microinstruction format, required a vertically-organized microinstruction. The problem of mutually-dependent fields was complicated and made the task of learning to microprogram using this evaluation board difficult. The background idea when considering this thesis topic was to remove the microprogrammer from the requirement to remember and control the various dependencies within the microword. The proposed system can be used by student microprogrammers, and the system will make the task of producing microroutines easy and secure.

LIST OF REFERENCES

1. Wilkes, M.V. and Stringer, J.B., "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer", Computer Structures: Principles and Examples, McGraw-Hill, 1982.

2. Husson, Samir S., Microprogramming: Principles and Practice, Prentice-Hall, 1970.

3. Patterson, David A., "Microprogramming".

4. Hamacher, V. Carl, Computer Organization, McGraw-Hill, 1978.

5. Hayes, John P., Computer Organization and Architechture, McGraw-Hill, 1978.

6. Salisbury, Alan B., Microprogrammable Computer Architectures, Elsevier, 1976.

7. Rauscher, Tomlinson G. and Adams, Phillip M., "Microprogramming: A Tutorial and Survey of Recent Developments", IEEE Transactions on Computers, Vol. c-29, 1 January 1980.

8. Patterson, David A., "An Experiment in High Level Language Microprogramming and Verification", Communications of the ACM, Vol. 24, October 1981.

9. Patterson, David A., Lew, Karl, and Tuck, Richard, "Towards an Efficient, Machine-Independent Language for Microprogramming", Transactions of the IEEE, 1979.

10. MacLennan, Bruce J., Principles of Programming Languages, Holt, Rinehart, and Winston, 1983.

11. White, Donnamaie E., Bit-Slice Design: Controllers and ALUs, Garland, 1981.

12. Hartrom, Thomas C., Lamont, Gary B., and Ross, Alan A., AMD AM29203 Evaluation Board User's Guide, Preliminary Draft, Advanced Micro Devices, 1983.

13. Advanced Micro Devices, Bipolar Microprocessor Logic and Interface Data Book, 1983.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	LtCol Alan A. Ross, Code 52Rs Naval Postgraduate School Monterey, California 93943	4
4.	LT Marcia E. Provance Fleet Intelligence Center, Pacific Pearl Harbor, Hawaii 96860	2
5.	Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, California 93943	1

Thesis
P9453
c.1

Provance
Implementation of a proposed system for automated microcode generation.